**AFRL-IF-RS-TR-2005-22**
**Final Technical Report**
**February 2005**

# SPECIFICATION-CARRYING SOFTWARE: EVOLVING SPECIFICATIONS FOR DYNAMIC SYSTEM COMPOSITION

**Kestrel Institute**

**Sponsored by**
**Defense Advanced Research Projects Agency**
**DARPA Order No. K506**

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

**The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.**

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**
**ROME RESEARCH SITE**
**ROME, NEW YORK**

## STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-22 has been reviewed and is approved for publication

APPROVED: /s/

JAMES M. NAGY
Project Engineer

FOR THE DIRECTOR: /s/

JAMES A. COLLINS, Acting Chief
Advanced Computing Division
Information Directorate

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 074-0188*

| | | |
|---|---|---|
| **1. AGENCY USE ONLY (Leave blank)** | **2. REPORT DATE**<br>FEBRUARY 2005 | **3. REPORT TYPE AND DATES COVERED**<br>Final Sep 00 – May 04 |

**4. TITLE AND SUBTITLE**
SPECIFICATION-CARRYING SOFTWARE: EVOLVING SPECIFICATIONS FOR DYNAMIC SYSTEM COMPOSITION

**5. FUNDING NUMBERS**
C - F30602-00-C-0209
PE - 62301E
PR - DASA
TA - 00
WU - 05

**6. AUTHOR(S)**
Matthias Anlauff,
Dusko Pavlovic,
and Douglas R. Smith

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Kestrel Institute
3260 Hillview Avenue
Palo Alto California 94304

**8. PERFORMING ORGANIZATION REPORT NUMBER**
N/A

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFT
525 Brooks Road
Rome New York 13441-4505

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
AFRL-IF-RS-TR-2005-22

**11. SUPPLEMENTARY NOTES**

AFRL Project Engineer: James M. Nagy/IFT/(315) 330-3173/ James.Nagy@rl.af.mil

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT *(Maximum 200 Words)***

EPOXI (Evolutionary Programming Over Explicit Interfaces) builds on the advanced mathematical foundation to enable the design and evolution of large-scale, heterogeneous, distributed, time-critical systems. The guiding philosophy of EPOXI is refinement of requirement specifications into code that is correct by construction. EPOXI emphasizes the support for design operations that establish or preserve required properties of the target system software. Refinement and coordination of changes to the software system were by means of formal change specifications, propagation of constraints through an architecture, gauges to measure component compliance and synthesis of glue-code to assure compliance/interoperability.

**14. SUBJECT TERMS**
Dynamic Software Assembly, Run-Time Software System Adaptation, Specification-Carrying Software, Spec Ware, EPOXI, Software Gauges

**15. NUMBER OF PAGES**
65

**16. PRICE CODE**

| | | | |
|---|---|---|---|
| **17. SECURITY CLASSIFICATION OF REPORT** | **18. SECURITY CLASSIFICATION OF THIS PAGE** | **19. SECURITY CLASSIFICATION OF ABSTRACT** | **20. LIMITATION OF ABSTRACT** |
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

# Contents

## List of Figures

# 1 Executive Summary

This report describes our research on the assured composability of software systems. Our project is based on the concept of *specification-carrying software* in which software artifacts carry with them all the information necessary to support composability and evolution. Without a representation of the properties and behavioral characteristics of a software component and its assumptions about its operating environment, it is impossible to insert a system into a context with any assurance.

We have developed a comprehensive mathematically well-founded framework for system specification, composition, development, and evolution. To support precise means for assessing composability of a system at design-time, our framework allows formal specification of (1) logical properties, (2) behavioral properties, (3) timing properties, and (4) environmental requirements. Specifications distinguish between the services offered by a system from the services required from its environment. Composability of a system requires that the system's environment provides the services that the system requires. A category of behavioral specifications provides the technical foundations of our approach. Specifications are the objects of the category and morphisms express part-whole relationships, refinement between machines, and the binding of required services to actual services. Composition is effected by the colimit operation, which is both semantically precise and scalably computable in near-linear time.

To assess composability, we developed techniques for formalizing (1) the compliance of a software artifact with its specification at several levels of granularity, and (2) the compliance of a component's environment with the services, behaviors, and properties that the component requires. The primary measure of composability is the existence of a morphism from each required-service specification of a system to the offered-service specifications of its environment. We also developed several techniques for ensuring composability in the presence of semantic mismatches between parts of a system. When there is a gap in the logical properties at an interface, we developed techniques for generating glue code when translation is possible. When there are race-condition-like timing problems between components, we developed constraint propagation techniques that eliminate uncoordinated behaviors while preserving logical properties.

The framework is partially implemented in the *EPOXI* system (Evolutionary Programming Over Explicit Interfaces). A variety of examples have been worked to explore the Epoxi concepts, including (1) a dynamic distributed sensor network from which Epoxi generated code running on a collection of Berkeley motes (demonstrated at Dasada Demo Days), (2) an avionic mission controller that is detailed in Section 4.2, (3) a concurrent garbage collection system [14], and (4) an embedded plant monitoring System [21].

The concepts of the Epoxi project have provided a foundation for a variety of other Kestrel projects, including projects on domain-specific generation of complex resource management systems, protocol composition and synthesis, embedded system design, and robustification of large-scale Java systems. To date, eight publications and two patent applications have directly resulted from this project.

# 2 Summary of Results

We briefly list the highlights of the research and development performed under this contract.

- *Evolving Specifications (Especs)* – We developed the concept of specification-carrying code via the formalism of evolving specifications [15, 17]. Especs formally specify the allowed behaviors of a process by means of logical/temporal properties, and by explicit modeling of states and transitions.

- *Espec Refinement* – We developed the concept of one Espec being a refinement of (or implementing) another Espec and this concept is defined by means of an Espec morphism. The morphism prescribes how logical properties translate, and how the concrete machine simulates the abstract machine.

- *Espec Composition* – We developed the concept of Espec composition by means of the colimit operation in the category of Especs. The colimit is efficiently computable and produces the Espec that specifies the (concurrent) composition of an arbitrary number of Especs (processes, machines, components).

- *Espec Parameters* – When modeling a component it is usually important to model its assumptions about its environment. A key insight of this project is that the environment model corresponds exactly to an Espec *parameter* to the component. That is, a component's requirements on its environment is specified by a parameter Espec. The guarantee that a component correctly provides its advertised services depends on the specification of its services together with the expectation that its environment implements the parameter Espec. Another aspect of parameters is that they refine contravariantly.

- *System Composition* – With the previous concepts we have, for the first time, a precise uniform foundation for system specification and composition. Each component is specified by Especs together with its parametric specification of environmental expectations. The interconnection of components is provided by a diagram that includes refinements from each component's parameter to its environment. The parameter is the gauge of compositionality. The measure of compliance is based on the correctness condition of a refinement – that all logical properties are preserved, and that the environment simulates the expected behaviors specified in the parameter.

- *Timing Properties* – We developed an extension of Especs that enables the specification of aspects of the physical world, including timing and other physical quantities. The result is a foundation for hybrid modeling of embedded systems. After composition it is necessary to refine the system to eliminate ill-timed behaviors. We developed a means for translating a composed Espec into a constraint satisfaction problem whose solution prescribes how to tighten the Espec to only allow coordinated behaviors between components.

- *Glue Code* – The services offered by one component may be expressed in slightly different data structures than the corresponding offered services of another component. To enable composition in such cases, we developed techniques for automatically generating glue code from the formal specification of the semantics of two services.

- *Epoxi* – We implemented Especs in the Epoxi system and performed a variety of demonstrations with it. For example, we demonstrated the ability to generate a dynamic architecture using Especs. A set of nodes (small low-cost, low power radio-linked computer/sensors called motes) are scattered randomly in the field. The basic system architecture comprises sensor components and radio connectors from each sensor to one of several data fusion components. When we attempt to refine the basic architecture to the motes, we find that the radio link has low reliability. To cope with such a fail-silent component/connector, a standard approach is to impose a heartbeat probe and a clock-based gauge that decides when the component/connector has failed. We formalized this heartbeat gauge architecture as an Espec diagram and formally composed it with the basic architecture to produce a dynamic adaptive sensor-net system. The Espec compiler translates the system to C and then into machine code for motes. The running system dynamically forms a network to flow sense data to the data fusion components, and dynamically adapts the network when a connector is gauged to have failed.

We briefly list the technical transition paths that we have been pursuing as an outgrowth of this project.

- *Planware* – The Especs foundation has been influential in a variety of ways. The Planware project for AFRL/Rome and the Electronic Systems Command, Hanscom AFB, uses Especs as the basis for precisely modeling scheduling problems. Users build Espec-based models of the tasks and resources of a scheduling application, then Planware automatically generates a branch-and-propagate algorithm. Under DARPA ANTS funding, Kestrel, Air Force Research Lab/Wright-Patterson AFB, and Lockheed are applying Planware to the modeling and generation of schedulers for handling time-critical targeting in Air Tasking Orders as part of the TBMCS (Theater Battle Management System). The current Planware model for this problem generates about 90,000 lines of CommonLisp (equivalent to about 270,000 lines of C) in a matter of minutes. The generated scheduler produces in a few minutes an air tasking order that is currently a full-time job for scores of officers in an Air Operations Center [3]. Furthermore, Kestrel and NSA have been discussing applying Planware to satellite scheduling.
- *Composition of Protocols* – A current project uses Especs in exploring dynamic game theory, system interaction, and formal analysis of protocols. Dramatic progress on the specification of protocols, a calculus for reasoning about them, and transformations for protocol composition are under active development [5]. Several new protocols with desirable properties have been derived within the formalism.

Here is a list of publications and patent applications that have directly resulted from this project.

1. Dusko Pavlovic and Douglas R. Smith, Composition and Refinement of Behavioral Specifications, Proceedings of the Sixteenth International Conference on Automated Software Engineering, IEEE Computer Society Press, Coronado Island, CA, 2001, 157-165.

2. Dusko Pavlovic and Douglas R. Smith, System Construction via Evolving Specifications, in Complex and Dynamic Systems Architectures (CDSA 2001), Brisbane, Australia, 2001.

3. Cordell Green, Dusko Pavlovic, and Douglas R. Smith, Software Productivity through Automation and Design Knowledge, Software Design and Productivity Workshop, Nashville TN, Dec 2001.

4. Dusko Pavlovic and Douglas R. Smith, Guarded Transitions in Evolving Specifications, in Proceedings of 9th International Conference on Algebraic Methodology And Software Technology (AMAST 2002), Eds. H. Kirchner and C. Ringeissen, Springer-Verlag LNCS 2422, September 9-13, 2002, St. Gilles les Bains, Reunion Island, France, 411-425.

5. Matthias Anlauff, Dusko Pavlovic, and Douglas R. Smith, Composition and Refinement of Evolving Specifications, invited paper in Proceedings of Workshop on Evolutionary Formal Software Development, July 2002, Copenhagen, Denmark.

6. Dusko Pavlovic and Douglas R. Smith, Software Development by Refinement, in *Formal Methods at the Crossroads: From Panacea to Foundational Support, UNU/IIST 10th Anniversary Colloquium*, Eds: B. Aichernig and T. Maibaum, Springer-Verlag LNCS 2757, 2003, 267-286.

7. Dusko Pavlovic, Peter Pepper, Douglas R. Smith, Colimits for Concurrent Collectors, *Verification: Theory and Practice: Festschrift for Zohar Manna*, N. Dershowitz (Ed.), Springer-Verlag LNCS 2772, 2003, 568–597.

8. Dusko Pavlovic and Douglas R. Smith, Evolving Specifications, in preparation, 2004.

9. D. Pavlovic, D.R. Smith, and S. Fitzpatrick, U.S. Patent Application 09/836582: Method and System for Self-Adaptive Code.

10. D. Pavlovic, D.R. Smith, and J. Liu, U.S. Patent Application 09/665179: Method and Apparatus for Determining Colimits of Hereditary Diagrams.

# 3 Foundations for System Composition

Originally, state machines were introduced and studied (by Turing, Moore, Mealy, and many others) as abstract, mathematical models of computers. More recently, though, software engineering tasks reached the levels where *practical* reasoning in terms of state machines has become indispensable: designing reactive, hybrid, embedded systems seems unthinkable without the various state modeling tools and languages, like Esterel, or Statecharts. Verifying high assurance systems by model checking is based on such state machine models. Moreover, one could argue that the whole discipline of object oriented programming is essentially a method for efficient management of state in software constructs.

However, there seems to be a conceptual gap between state machines as theoretical versus practical devices. A notable effort towards bridging this gap are Gurevich's Abstract State Machines [8]: on one hand, they are a foundational paradigm of computation, explicitly compared with Turing machines; on the other hand, they have been used to present practically useful programming languages, capturing semantical features of C, Java, and others. However, the absence of powerful typing and structuring (abstraction, encapsulation, composition. . . ) mechanisms makes them unsuitable for development and management of large software systems.

We wish to investigate a representation of state machines in a framework for large-scale software specification development ("from-specs-to-code"). Previous work at Kestrel Institute has implemented the Specware/Designware framework for the development of functional programs that is based on a category of higher-order logical specifications, composition by colimit, and refinement by diagram morphisms [24, 20]. The current work builds on and extends this framework with behavioral specifications (Especs), representing state machines as diagrams of specifications, and again using composition by colimit and refinement by diagram morphism. Related approaches to representing behavior in terms of a category of specifications include [6, 10].

The goal is to build a *practical* software development tool, geared towards large, complex systems, with reactive, distributed, hybrid, embedded features, and with high assurance, performance, reliability, or security requirements, all on a clean and simple semantical foundation.

## 3.1 Specifications

A *specification* is a finite presentation of a theory. The signature of a specification provides the vocabulary for describing objects, operations, and properties in some domain of interest, and the axioms constrain the meaning of the symbols. The theory of the domain is the closure of the axioms under the rules of inference. Although the examples in this paper are drawn from first-order theories, all of the constructions are institution-independent [7], mainly depending on existence of colimits and some limits.

A *specification morphism* (or *interpretation* translates the language of one specification into the language of another specification, preserving the property of provability, so that any theorem in the source specification remains a theorem under translation. Formally, a specification morphism $m : T \to T'$ is given by a map from the sort and operator symbols of the *domain* spec $T$ to the symbols of the *codomain* spec $T'$. An interpretation is the slightly more general case where a symbol may map to an expression. When there is no point in distinguishing them we use the generic term morphism. To be a morphism also requires that every axiom of $T$ translates to a theorem of $T'$. It then follows that a morphism translates theorems of the domain specification to theorems of the codomain.

**Starting point: category of specs** Why do we begin from a specification framework? Because that is the formal, semantical embodiment of the actual engineering practice: the development from specs to code, and the evolution of code and specs together.

Why do we arrange the specification framework in a category? Specifications are just *representations* of the design knowledge they formalize, just like programs are just particular representations of the computations that run when they are executed. Each piece of software can be represented by many specifications, just like it can be implemented by many different programs. — *The different specifications of the same structure are connected by the interpretations.* They perform often complex tasks of renaming signatures, while preserving the truth. Specifications and morphisms form a category.

## 3.2 Evolving Specifications

There are four key ideas underlying our representation of state machines as evolving specifications (Especs). Together they reveal an intimate connection between behavior and the category of logical specifications. The first three are explicit in Abstract State Machines [8].

1. *A state is a model* – A state of computation can be viewed as a snapshot of the abstract computer performing the computation. The state has a set of named stores with values that have certain properties.

2. *A state transition is a finite model change* – A transition rewrites the stored values in the state.

3. *An abstract state is a theory* – Not all properties of a state are relevant, and it is common to group states into abstract states that are models of a theory. The theory provides the structure (sorts, variables, operations), plus the axioms that describe common properties (i.e. invariants).

4. *An abstract transition is a theory morphism* – Just as a class of states/models may be described by a theory, a class of transitions may be described by an appropriate theory morphism. As a simple case, consider the correctness of an assignment statement relative to a precondition $P$ and a postcondition $Q$; i.e. a Hoare triple $P\ \{x\ :=\ e\}\ Q$. If we consider the initial and final states as characterized by theories $\mathsf{thy}_{pre}$ and $\mathsf{thy}_{post}$ with theorems $P$ and $Q$ respectively, then the triple is valid iff $Q[e/x]$ is a theorem in $\mathsf{thy}_{pre}$. That is, the triple is valid iff the symbol map $\{\ x\ \mapsto\ e\}$ induces a interpretation from $\mathsf{thy}_{post}$ to $\mathsf{thy}_{pre}$. Note that interpretation goes in the *opposite* direction from the state transition.

> More generally, a transition may have a guard that controls its application. Correspondingly we define (in Section 3.3) the notion of a guarded interpretation, which is essentially an interpretation into the extension of the codomain theory with the guard as an extra axiom. Specs with guarded interpretations form a category $\mathsf{Spec}_{\vdash}$.

*The basic idea of Especs is to use specifications as state descriptions, and to use guarded interpretations to represent transitions between state descriptions.*

The idea that abstract states and abstract transitions correspond to specs and guarded interpretations suggests that state machines are diagrams over $\mathsf{Spec}_{\vdash}^{op}$.

**Remark.** *What is static about state?* Note that the term "abstract state", although consistent with current convention, is nevertheless misleading. While a (concrete) state as a first-order structure is truly static, our intention for abstract states is much broader. What is static about an abstract state is the signature and axioms. A great deal of state change can happen within a single abstract state. Looking ahead, an abstract state/specification can naturally include continuous variables that model physical processes, together with constraints on their values and derivatives. The "static" theory then denotes a mixture of continuous flow and discrete variable change. Again, *what is static about abstract state is the invariant structure and properties expressed by its logical specification.* This view lets us naturally extend the expressiveness of Especs to hybrid systems, and further to nonfunctional properties of a system (such as timing and constraints on resource utilization). In this sense an Espec is really an activity machine (with nodes representing activities), but we continue to use the tradition term "state machine". However, abstract states in Especs are typically named with gerunds, suggesting that they correspond to activities. In this view abstract transitions also have more import – they correspond to a change of invariant mathematical structure (hence "evolving specifications").

The specification of each state description corresponds to its local structure and properties/invariants. The specification common to all state descriptions specifies the global structure and invariants of the system. Any structure that is common to all states that a computation can reach is formalized as a (global) specification; the common structure includes variables and their sorts, as well as axioms (global invariants) and operations (global constants).

### 3.3 The Category of Guarded Interpretations

The foundation for evolving specifications is the category of specifications with guarded interpretations as morphisms.

Let $K$ and $L$ be two states and $K \xrightarrow{\Phi \vdash f} L$ a transition, consisting of the guard, i.e. a predicate $\Phi$, and the action, i.e. update $f$. Intuitively, it can be understood as the command `if` $\Phi$ `then` $f$, executed at the state $K$, and leading into the state $L$ by $f$ whenever the condition $\Phi$ is satisfied. More precisely, it is executed in two steps:

- at the state $K$, the condition $\Phi$ is evaluated;
- if it is satisfied, the action $f$ is performed, leading to the state $L$.

The action $f$ is assumed to *rewrite* the signature of the state description $L$ in terms of the signature of the state description $K$. Such rewriting can be viewed as a set of *assignments*, possibly to higher-order variables. Such assignments are often called signature *updates*, especially in the Abstract State Machines community.

*A transition* $K \xrightarrow{\Phi\vdash f} L$ *thus boils down to an interpretation* $L \xrightarrow{f} K \wedge \Gamma$*, where states* $K$ *and* $L$ *are identified with logical theories describing them.*

The meaning of this is as follows. A computation is in state $K$ if the program variables, operations and relations form a model of the state description of $K$. If we are thus given a model of $K$, and an interpretation $L \longrightarrow K$, then we can of course assign to the signature of $L$ meanings along this interpretation, and thus reinterpret the model of $K$ as a model of $L$, and arrive at the state $L$.

However, if such transition is *only* to be made when the guard $\Phi$ is satisfied, then we have

$$K \twoheadrightarrow \; K \;\&\; \Phi \xleftarrow{f} \; L$$

where we abuse notation in letting $K \wedge \Phi$ denote the spec obtained by adding axiom $\Phi$ to spec $K$ (assuming $\Phi \in \mathcal{L}_K$). The left-hand side interpretation is the trivial, identity map on the signature of $K$, just adjoining to its theory the axiom $\Phi$.

Opspans like this will be our general representation of guarded transitions, and will be denoted

$$K \xrightarrow{\quad \Phi\vdash f \quad} M$$

Sequential composition of guarded transitions is given by the rule

$$\frac{K \xrightarrow{\Phi\vdash f} L \quad L \xrightarrow{\Gamma\vdash g} M}{K \xrightarrow{\Phi\wedge f(\Gamma)\vdash f\cdot g} M}$$

or diagrammatically

$$\begin{tikzcd}
K \arrow[d] & L \arrow[dl, "f"'] \arrow[d] & M \arrow[dl, "g"] \\
K \wedge \Phi \arrow[d] & L \wedge \Gamma \arrow[dl, "f"] & \\
K \wedge \Phi \wedge f(\Gamma) & &
\end{tikzcd}$$

The category $\mathsf{Spec}_\vdash$ of specifications and guarded transitions will have specifications as its objects, while the hom-sets will be

$$\mathsf{Spec}_\vdash(K,L) \quad = \quad \{(\Phi \vdash f) \mid \Phi \in \mathcal{L}_K \text{ and } f \in \mathsf{Spec}(L, K\wedge\Phi)\}$$

**Remark.** The construction of $\mathsf{Spec}_{\vdash}$ is an instance of a more general construction in which morphisms are treated as a combination of arrows from a factorization system [17]. Here the guard can be treated as an abstract monic and the interpretation can be treated as an abstract epi.

**Remark.** The computational work of this transition takes place in state $K$. The guard is evaluated in $K$, as are the computations of the right-hand sides of the parallel assignments. On the other hand, the evaluation of which variables to update (e.g. calculating the array offset in `A[i] := e`) is charged to state $L$.

### 3.4 The Slice Category of Guarded Interpretations

In general, the state descriptions of a machine are not unrelated specs, but they share vocabulary, definitions, and properties that capture whatever may be known about the global invariants and the intent of the program. The universe from which the states are drawn is thus the category $A/\mathsf{Spec}_{\vdash}$, of all specs inheriting global spec $A$, rather than just $\mathsf{Spec}_{\vdash}$.

The abstract states $K$ and $L$ thus come with the morphisms/interpretations $k : A \longrightarrow K$ and $\ell : A \longrightarrow L$ of the globally visible signature and the invariants specified in $A$. [1]

**Definition 3.1** *For a fixed spec $A$, the category $\mathsf{ext}_A = A/\mathsf{Spec}_{\vdash}$ consists of*

**objects:** *extensions $A \xrightarrow{k} K$ of the spec $A$ in the category $\mathsf{Spec}$;*

**arrows:** *given $A \xrightarrow{k} K$ and $A \xrightarrow{\ell} L$, a morphism $(\Phi \vdash f) : k \longrightarrow \ell$ is a guarded transition*

$$\begin{tikzcd} & A \arrow[dl, "k"'] \arrow[dr, "\ell"] & \\ K \arrow[rr, "\Phi\vdash f"'] & & L \end{tikzcd}$$

*which means*

$$\begin{tikzcd} & & A \arrow[dll, "k"'] \arrow[drr, "\ell"] & & \\ K \arrow[rr, two heads] & & K \wedge \Phi & & L \arrow[ll, "f"'] \end{tikzcd}$$

The hom-sets are thus

$$\mathsf{ext}_A(k,\ell) \quad = \quad \{(\Phi \vdash f)\ |\Phi \in \mathcal{L}_K \text{ and } f \in \mathsf{Spec}(L, K \wedge \Phi) \text{ and } k \cdot i = \ell \cdot f\}$$

where $i : K \longrightarrow K \wedge \Phi$ is the identity signature map, just strengthening the theory $K$.

[1] While $\mathsf{Spec}$ is fibered over $\mathsf{Lang}$ by the functor mapping each spec $K$ to its language $\mathcal{L}_K$, the category $A/\mathsf{Spec}$ is fibered over $\mathcal{L}_A/\mathit{Lang}$, mapping each interpretation $A \longrightarrow K$ to the underlying language interpretation.

**Remark.** Note that the construction of $\mathsf{Spec}_{\vdash}$ is just a fibered form of adjoining objects as guards: for each language $\Sigma$ the semilattice $\langle|\mathsf{Spec}_\Sigma|, \wedge, \top\rangle$ of all theories over $\Sigma$ is adjoined to the fiber $\mathsf{Spec}_\Sigma$ as the monoid of guards.

The construction $\mathsf{ext}_A = A/\mathsf{Spec}_{\vdash}$ is, on the other hand, a fibered form of the slightly more general construction. The category $A/\mathsf{Spec}$ is fibered over the category $\mathcal{L}_A/\mathsf{Lang}$ of languages interpreting $\mathcal{L}_A$. But this time, each fiber $(A/\mathsf{Spec})_\sigma$ over $\sigma : \mathcal{L}_A \longrightarrow \Sigma$ is assigned not its own monoid of objects, but again just the semilattice of theories over $\Sigma$.

### 3.5 Especs Defined

The concept of Espec is now formally defined.

**Definition 3.2** *A* graph *s consists of two sets* $\mathsf{edge}_s$ *and* $\mathsf{node}_s$*, and two functions,* $\mathsf{dom}_s$ *and* $\mathsf{cod}_s$ *from* $\mathsf{edge}_s$ *to* $\mathsf{node}_s$*.*

*A* shape *is a graph s that is*

- *reflexive, in the sense that there is a function* $\mathrm{id}_s : \mathsf{node}_s \longrightarrow \mathsf{edge}_s$*, which assigns a distinguished loop to each node;*
- *distinguished initial node $i$, and a set $O$ of final nodes $o$;*

*Together with the morphisms preserving all displayed structure, shapes form the category* $\mathsf{Shape}$.

**Definition 3.3** *An* evolving spec*, or* **Espec** *A consists of*

- *a logical theory* $\mathsf{spec}_A$*, and*
- *a state machine* $\mathsf{prog}_A$*, presented by*
  - *a reflexive graph* $\mathsf{shape}_A$*;*
  - *a morphism of reflexive graphs* $\mathsf{st}_A : \mathsf{shape}_A \longrightarrow \mathsf{ext}_A$

**Notation and terminology.** If $n$ is a node of $\mathsf{shape}_A$, the codomain of $\mathsf{st}_A(n)$ is written as $\mathsf{mode}_A(n)$. If $u : m \to n$ is an edge of $\mathsf{shape}_A$, its image $\mathsf{st}_A u$ is usually written as $\mathsf{step}_A u : \mathsf{mode}_A m \twoheadrightarrow \mathsf{mode}_A n$ .

As a $\mathsf{ext}_A$-arrow, $\mathsf{step}(u)$ consists of two pieces of data:

- a formula $\mathsf{guard}(u)$ in the language of $\mathsf{mode}(m)$, and
- an interpretation $\mathsf{assign}(u) : mode(n) \longrightarrow \mathsf{mode}(m) \wedge \mathsf{guard}(u)$

In summary,

- $\mathsf{mode}$ assigns to each $\mathsf{shape}$-node $n$ a *state description* $\mathsf{mode}(n)$, which comes with an interpretation $\mathsf{st}_A(n) : \mathsf{spec}_A \longrightarrow \mathsf{mode}(n)$;
- $\mathsf{step}$ assigns to each $\mathsf{shape}$-edge $u : m \rightarrow n$ a *step* (or *transition*) $\mathsf{step}(u) :\ \mathsf{mode}(m) \longrightarrow \mathsf{mode}(n)$ , keeping $\mathsf{spec}_A$ invariant, in the sense that

$$\begin{tikzcd}
 & \mathsf{spec}_A \arrow[dl, "\mathsf{st}(m)"'] \arrow[dr, "\mathsf{st}(n)"] & \\
\mathsf{mode}(m) \arrow[rr, "\mathsf{step}(u)"'] & & \mathsf{mode}(n)
\end{tikzcd}$$

i.e.

$$\begin{tikzcd}
 & \mathsf{spec}_A \arrow[dl, "\mathsf{st}(m)"'] \arrow[dr, "\mathsf{st}(n)"] & \\
\mathsf{mode}(m) \arrow[dr] & & \mathsf{mode}(n) \arrow[dl, "\mathsf{assign}(u)"] \\
 & \mathsf{guard}(u)\ \&\ \mathsf{mode}(m) &
\end{tikzcd}$$

## Espec Syntax and Examples

Here is a brief introduction to the syntax of Especs and Espec morphisms. By way of illustration, we specify the problem of computing the greatest-common-divisor of two positive numbers.

Semantically, an Espec denotes a set of behaviors. This, together with the hierarchical nature of Especs - modes may be Especs, implies that each mode denotes a set of behaviors. This semantic is consistent with the restriction of refinement to modes (there is no notion of transition refinement in Especs).

Especs can be viewed as proper extensions of ordinary higher-order specifications. So we first review the general syntax of a Specware spec (details may be found in the documentation on Specware at www.specware.org). The general form of a spec is

```
S = spec
     imports
     sorts
     ops
     axioms
     theorems
    end-spec
```

where

imports: A spec may import any number of specifications.

sorts: A spec may introduce and define any number of sort (type) symbols.

ops: A spec may introduce and define any number of operation symbols.

axioms: The axioms are expressed in terms of the imported and introduced sorts and operations, using the built-in syntax of higher-order logic.

theorems: The theorems are consequences of the axioms.

For example, GCD-base is a simple specification that imports a specification of the Integers, defines the (sub)sort of positive integers, introduces a new function GCD and constraints its possible meanings by the `gcd-spec` axiom. Finally, a simple theorem is presented, which follows from `gcd-spec` and integer theory. The purpose of this spec is to specify the GCD problem without giving a computable definition of it.

```
GCD-base = espec
    import Integer
    sort Pos = (Integer|positive?)
    def positive?(i:Integer):boolean = (i > 0)
    op gcd : Pos, Pos -> Pos
    axiom gcd-spec is          ;; this axiom specifies the gcd problem
        gcd(x,y) = z
          => (divides(z,x) & divides(z,y)
             & forall(w:Pos)(divides(w,x) & divides(w,y) => w <= z))
    theorem gcd(x,x) = x
 end-espec
```

The general form of an Espec is

```
M = espec
     parameter
     value
     import
     assumes
     ensures

     vars
     ops
     axioms

     events
     procs
     modes
     steps
    end-espec
```

where the `imports`, `sorts`, `ops`, `axioms/theorems` clauses are the same as in specs, and

parameter: An Espec may have any number of parameters, which are themselves Especs. The role of parameters is discussed more fully in Section 3.8.

value: The name and type of the output value.

assumes: The assumes clause is a logical constraint on the variables in the parameters that specifies properties of the initial state of any behavior of the Espec (sometimes called the precondition).

ensures: The ensures clause is a logical constraint on the variables in the parameters together with the value that specifies properties of the final state of any behavior of the Espec relative to values of variables in the initial state (sometimes called the postcondition).

vars: The vars are variables (symbols that may have different associated values in different states).

events: The events represent synchronized transitions with the environment. An event may represent message-passing, control signals, or simply a named transition. Unlike procedures, the controlling entity of an event does not expect or wait for a return of control. An event has a type that constrains its content.

procs: The procs are Especs that represent local procedures (or methods).

modes: The modes are Especs that specify abstract states of the Especs.

steps: The steps specify abstract transitions of the Especs. They correspond to guarded interpretations.

For example, GCD-0 is a simple Espec that specifies a procedure for computing GCD. It imports GCD-base and takes two data parameters (note that the parameter is a trivial Espec, having only vars and no modes or steps). GCD-0 introduces the output variable `Z` and specifies that in any behavior of GCD-0, the value of `Z` in the final state will be the greatest common divisor of the input values `X-in` and `Y-in`. Finally, GCD-0 introduces two modes and a step that specify the abstract behavior of the computation - the single transition has the effect of storing the gcd of `X-in` and `Y-in` into `Z`. Note that this is still an underspecified computation, since the `gcd` function has no definition. We will refine this simple Espec to a computable Espec later. The naming of modes and steps is needed to support the refinement of Especs.

```
GCD-0 = espec
  import GCD-base
  parameters {X-in: Pos, Y-in : Pos}
  value Z : Pos
  assumes true
  ensures Z = gcd(X-in,Y-in)

  One = init mode end-mode

  Two = fin mode    ;; this mode extends the global spec with a local axiom
    axiom Z = gcd(X-in,Y-in)
  end-mode
  Out = step : One -> Two   ;; transition from mode One to mode Two
     Z := gcd(X-in,Y-in)
  end-step
end-espec
```

A slightly more compact presentation folds each step into the mode from which it originates.

```
GCD-0 = espec
  import GCD-base
  parameters {X-in: Pos, Y-in : Pos}
  value Z : Pos
  assumes true
  ensures Z = gcd(X-in,Y-in)

  One = init mode
         Out = step -> Two
                Z := gcd(X-in,Y-in)
               end-step
        end-mode

  Two = fin mode
    axiom Z = gcd(X-in,Y-in)
  end-mode

end-espec
```

Espec GCD-1, below, refines GCD-0. The state machine expresses the classical GCD algorithm, which might have been generated by a design tactic. GCD-1 extends the logical spec of GCD-0 with two local variables X and Y. Essentially, the refinement adds a new mode and two looping transitions that preserve the key loop invariant of the program: X and Y change under the transitions, but always so that their GCD is the same as the GCD of the input values X-in and Y-in.

```
GCD-1 = espec
  import GCD-base
  parameters {X-in: Pos, Y-in : Pos}
  var X,Y : Pos

  One = init mode end-mode

  Loop = mode
    axiom gcd(X-in,Y-in) = gcd(X,Y) ;; loop invariant
  end-mode

  Two = mode
    axiom Z = X
    axiom X = Y
    axiom Z = gcd(X-in,Y-in)
  end-mode

  initialize = step : One -> Loop
    X := X-in
    Y := Y-in
  end-step

  Loop1 = step : Loop -> Loop
    X>Y -> X := X - Y
  end-step

  Loop2 = step : Loop -> Loop
    Y>X -> Y := Y - X
  end-step

  Out = step : Loop -> Two
    X=Y -> Z := X
  end-step
end-espec
```

or more compactly

```
GCD-1 = espec
  import GCD-base
  parameters {X-in: Pos, Y-in : Pos}
  var X,Y : Pos

  One = init mode
    step : One -> Loop
      X := X-in
      Y := Y-in
      end-step
    end-mode

  Loop = mode
    axiom gcd(X-in,Y-in) = gcd(X,Y)    ;; loop invariant
    Loop1 : step -> Loop
      X>Y -> X := X - Y
      end-step
    Loop2 : Loop -> Loop
      Y>X -> Y := Y - X
      end-step
    Out : Loop -> Two
      X=Y -> Z := X
      end-step
   end-mode

  Two = mode
    axiom Z = X
    axiom X = Y
    axiom Z = gcd(X-in,Y-in)
  end-mode
end-espec
```

It is straightforward to check that `GCD-1` is internally consistent; e.g. to show that `Loop1` corresponds to a translation, we must show

$$\mathit{Loop},\ X > Y \ \vdash\ \mathit{gcd}(X\text{–}\mathit{in}, Y\text{–}\mathit{in}) \ = \ \mathit{gcd}(X - Y, Y)$$

The syntax, semantics, and correctness conditions of refinements are addressed in Section 4.

## 3.6 Refinements

We now define the concept of a refinement (or morphism) between two Especs. A characteristic of Espec refinements is that logical structure and behavior refine contravariantly. If $A$ refines to $B$, then the spec of $A$ refines to the spec of $B$ by an interpretation, but the program (or state machine) of $B$ maps into the program of $A$, simulating it. So a refinement preserves the logical structure of $A$ in $B$ and preserves the behavior of $B$ in $A$.

**Definition 3.4** *Given Especs $A$ and $B$, a refinement $f : A \longrightarrow B$ consists of:*

$$\begin{tikzcd}
A \arrow[d, "f"'] & = & \langle \mathsf{spec}_A, \arrow[d, "f_{\mathsf{spec}}"'] & \mathsf{shape}_A \arrow[r, "\mathsf{st}_A"] \arrow[dr, Rightarrow, "f_{\mathsf{mode}}"] & \mathsf{ext}_A^{op} \rangle \\
B & = & \langle \mathsf{spec}_B, & \mathsf{shape}_B \arrow[r, "\mathsf{st}_B"'] \arrow[u, "f_{\mathsf{shape}}"'] & \mathsf{ext}_B^{op} \rangle \arrow[u, "f^*_{\mathsf{spec}}"']
\end{tikzcd}$$

- *a* structure map *(or interpretation)* $f_{\mathsf{spec}}$
- *a* behavior map *(or simulation)* $f_{\mathsf{prog}} = \langle f_{\mathsf{shape}}, f_{\mathsf{mode}} \rangle$*, where*
  - $f_{\mathsf{shape}}$ *is a reflexive graph morphism, preserving the initial and the final nodes,*
  - $f_{\mathsf{mode}}$ *is* $\mathsf{spec}_A$*-preserving natural transformation; this naturality and preservation amount to the commutativity of Figure 1 for every $v : k \to \ell$ in* $\mathsf{shape}_B$ *(see notes below).*
  - *Together,* $f_{\mathsf{spec}}$ *and* $f_{\mathsf{prog}}$ *must also satisfy the* guard condition*: for every edge $v : k \to \ell$ in* $\mathsf{shape}_B$ *and edge $u = f_{shape}(v)$ in* $\mathsf{shape}_A$ *and every edge $u = f_{\mathsf{shape}}(k) \to n$ in* $\mathsf{shape}_A$*,*

$$f_{\mathsf{spec}}(\mathsf{guard}_A(u)) \iff \bigvee_{\substack{f_{\mathsf{shape}}(v)=u \\ \mathsf{dom}(v)=k}} \mathsf{guard}_B(v)$$

- *The inverse-image functor $f^*_{\mathsf{spec}}$ acts on the category of extensions: $f^*_{\mathsf{spec}}(\mathsf{st}(n)) = \mathsf{st}(n) \circ f_{spec}$.*

Clearly, Especs and refinements form a category, which we shall denote $\mathsf{ESpec}$.

**Intuition.** The last diagram tells that the components of $f_{\mathsf{mode}}$ coherently extend $f_{\mathsf{spec}}$ from the global specs $\mathsf{spec}_A$ and $\mathsf{spec}_B$ to their extensions $\mathsf{mode}_A$ and $\mathsf{mode}_B$. Just like $\mathsf{spec}_B$ refines $\mathsf{spec}_A$ because it proves all formulas in the image $f_{\mathsf{spec}}[\mathsf{spec}_A]$, each $\mathsf{mode}_B(n)$ refines $\mathsf{mode}_A(f_{\mathsf{shape}}(n))$ because it proves all formulas in the image $f_{\mathsf{mode}}(n)[\mathsf{mode}_A(f_{\mathsf{shape}}(n))]$. The *structural* refinement is thus extended from $f_{\mathsf{spec}} : \mathsf{spec}_A \longrightarrow \mathsf{spec}_B$ to $f_{\mathsf{mode}} : \mathsf{mode}_A \longrightarrow \mathsf{mode}_B$. Its naturality ensures that each transition $\mathsf{step}_B(v)$ of $B$ extends the transition $\mathsf{step}_A(f_{\mathsf{shape}}(v))$ of $A$.

The guard condition ensures that every behavior of $B$ maps to a behavior of $A$. There are stronger versions of the guard condition that also ensure that $B$ simulates all of $A$'s behaviors, and weaker versions that support reduction of nondeterminism.

$$\begin{tikzcd}
\mathsf{mode}_A(f_{\mathsf{shape}}(k)) \arrow[r, hook] \arrow[ddd, "f_{\mathsf{mode}}(k)"'] & \mathsf{mode}_A(f_{\mathsf{shape}}(k)) \wedge \mathsf{guard}_A\,(f_{\mathsf{shape}}(v)) \arrow[ddd, "f_{\mathsf{mode}}(k)"'] & & \mathsf{mode}_A(f_{\mathsf{shape}}(\ell)) \arrow[ll, "\mathsf{assign}_A(f_{\mathsf{shape}}(v))"'] \arrow[lll, bend right, "\mathsf{step}_A(f_{\mathsf{shape}}(v))"'] \arrow[ddd, "f_{\mathsf{mode}}(\ell)"] \\
& & \mathsf{spec}_A \arrow[llu, "\mathsf{st}_A(f_{\mathsf{shape}}(k))"] \arrow[ru, "\mathsf{st}_A(f_{\mathsf{shape}}(\ell))"'] \arrow[d, "f_{\mathsf{spec}}"] & \\
& & \mathsf{spec}_B \arrow[lld, "\mathsf{st}_B(k)"'] \arrow[rd, "\mathsf{st}_B(\ell)"] & \\
\mathsf{mode}_B(k) \arrow[r, two heads] & \mathsf{mode}_B(k) \wedge \mathsf{guard}_B(v) & & \mathsf{mode}_B(\ell) \arrow[ll, "\mathsf{assign}_B(v)"] \arrow[lll, bend left, "\mathsf{step}_B(v)"]
\end{tikzcd}$$

Figure 1: **Naturality Condition of a Refinement**

**Proposition:** Every behavior of $B$ is a behavior of $A$.

Proof: Intuitively, the commutativity requirement of the natural transformation implies the commutativity of the underlying models/states. From this it follows that for every state that $B$ can be in and enabled transition, there is a corresponding state of $A$ such that the corresponding transition of $A$ is enabled. Coinductively then, every behavior of $B$ maps to a behavior of $A$.

But is *every* transition of $A$ extended in $B$? Yes, insofar as its guard is ever enabled. This is ensured by the guard condition, which, in our context, says that the behavior $\mathsf{st}_B$ can simulate $\mathsf{st}_A$. More precisely, the inverse graph of $f_{\mathsf{shape}}$ is a simulation in the usual sense of concurrency theory: if $f_{\mathsf{shape}}(k) = m$ and there is a transition $u$ out of $m$, enabled in a given run, in the sense that $\mathsf{guard}_A(u)$ is true, then there must be an enabled transition $v$ out of $k$ in $B$, with $f_{\mathsf{shape}}(v) = u$. The other way around, whenever some $v$, $f_{\mathsf{shape}}(v) = u$ is enabled, $u$ must be enabled.

Note that the guard condition implies that $f_{\mathsf{shape}}$ must be surjective, at least as far as $B$ can tell through $f_{\mathsf{spec}}$: namely, $f^{-1}_{\mathsf{shape}}(u)$ can be empty only if the guard $\mathsf{guard}_A(u)$, interpreted by $B$ as $f_{\mathsf{spec}}(\mathsf{guard}_A(u))$, is always false.

Note that every mode has the identity self-transition, sometimes called a stuttering step, whose action is minimal. The guard of the identity is true. The guard condition requires that the refinement of the identity must never deadlock.

The presentation of an Espec refinement `R` from `A` to `B` has the following general form:

```
R = morphism A -> B
     parameter  parameter-morphism
     commonspec commonspec-morphism
     structure  structure-map
```

where

parameter: The parameter-morphism is a contravariant morphism from the parameter of B to the parameter of A.

commonspec: The commonspec-morphism is a covariant specification morphism from the common spec of A to the common spec of B.

structure: The structure-map is a contravariant map from the modes of B to the modes of A and the steps of B to the steps of A. The structure-map induces the natural transformation of R.

**Example.** Let us return to the example in Section 2. The refinement from `GCD-0` to `GCD-1` can be written

```
GCD-ref = morphism GCD-0 -> GCD-1
            parameter  {X_in +-> X_in,
                        Y_in +-> Y_in}
            commonspec {gcd +-> gcd,
                        Z   +-> Z }
            structure  {One        +-> One,
                        Loop       +-> One,
                        Two        +-> Two,
                        Initialize +-> Id(One),  % Identity arrow on a Mode
                        Loop1      +-> Id(One),
                        Loop2      +-> Id(One),
                        Out        +-> Out }
```

where `+->` is the ascii form of the maps-to symbol ($\mapsto$).

Three of the steps map to the identity step on mode One in `GCD-0` because they only change the local variables `X` and `Y`, corresponding to identity steps in `GCD-0` (these are sometimes called stuttering steps). Checking the components of the natural transformation is straightforward – the proof obligations include showing that $f_{\mathsf{mode}}(k)$ is a translation for all nodes $k$ in $\mathsf{shape}_A$; e.g. that the axioms of mode `One` in `GCD-0` translate to theorems in mode `Loop` in `GCD-1`. Checking the guard condition is also straightforward; e.g. for step `Loop1` in `GCD-1`, the guard condition instantiates to

$$Loop \vdash\ X > Y \implies true$$

where the consequent is the guard on step $id_{One}$ in `GCD-0`.

Epoxi does not require the statement of identity symbol translations, so `GCD-ref` can be presented more compactly as

```
GCD-ref = morphism GCD-0 -> GCD-1
            parameter  {}
            commonspec {}
            structure  {Loop       +-> One,
                        Initialize +-> Id(One),
                        Loop1      +-> Id(One),
                        Loop2      +-> Id(One) }
```

## 3.7 Colimits

The ordinary comma categories are well known [11, ch. II.6]. Given the functors $G : \mathcal{X} \longrightarrow \mathbb{C}$ and $H : \mathcal{Y} \longrightarrow \mathbb{C}$ the comma category[2] $G/H$ will consist of the triples $\langle X, GX \xrightarrow{c} HY, Y\rangle$ as objects, where $X$ is an object of $\mathcal{X}$, $Y$ an object of $\mathcal{Y}$, and $c$ an arrow of $\mathcal{C}$. Given two such objects, $A$ and $B$, an arrow $f : A \longrightarrow B$ is a pair $\langle f_X, f_Y\rangle$, making the following square commute

$$\begin{tikzcd}
A \arrow[d, "f"'] & = & \langle X_A, \arrow[d, "f_X"'] & GX_A \arrow[r, "c_A"] \arrow[d, "Gf_X"'] & HY_A, \arrow[d, "Hf_Y"] & Y_A\rangle \arrow[d, "f_Y"] \\
B & = & \langle X_B, & GX_B \arrow[r, "c_B"'] & HY_B, & Y_B\rangle
\end{tikzcd}$$

The *lax* comma category $G//H$ can be made when $\mathbb{C}$ is a 2-category. The commutativity requirement on the square in the above diagram can then be relaxed to commutativity up to a 2-cell.

Towards a representation of ESpec, consider the functors

- $D : \mathsf{Shape} \longrightarrow \mathsf{Cat}$, mapping each shape to the free category over the underlying reflexive graph, and
- $M : \mathsf{Spec}^{op} \longrightarrow \mathsf{Cat}$, mapping each spec to the category $(\mathsf{spec}/\mathsf{Spec})^{op}$, and each spec morphism to the functor induced by precomposition.[3]

By projecting away the guards, both from Especs and from their refinements, we get the functor $\mathsf{ESpec} \longrightarrow D//M$.

The point is that this functor creates colimits. And both limits and colimits in comma categories, even lax, are constructed in a standard way. A diagram in $G/H$, of course, consists of a diagram in $\mathcal{X}$ and corresponding one in $\mathcal{Y}$ — projected into $\mathcal{C}$ by $G$ and $H$ respectively, and connected by the suitable commutativity conditions there. The (co)limit in $G/H$ is obtained by separately constructing the two (co)limits in $\mathcal{X}$ and $\mathcal{Y}$, and by projecting and connecting them up in $\mathcal{C}$. In the lax case, only this last step gets more complicated.

The colimits in ESpec will thus be constructed from the colimits in Spec, the limits in Shape, plus some wiring to connect them in Cat.

[2] The notation $G/H$ is not standard, but is justified by the fact that the comma construction subsumes the slice categories, besides the arrow categories and their derivatives.

[3] This functor can be viewed as a generalization of the functor $\mathsf{Mod} : \mathsf{Spec}^{op} \longrightarrow Cat$.

First of all, recall that all colimits can be derived from the initial object and the pushouts. Of course, the initial Espec consists of the empty spec, and a one-state-one-step program (with the state represented by the empty spec).

To describe the pushout of Especs, suppose we are given a span of Especs

$$\begin{tikzcd}
& \mathsf{spec}_A \arrow[dl, "f_{\mathsf{spec}}"'] \arrow[dr, "g_{\mathsf{spec}}"] & \\
\mathsf{spec}_B & & \mathsf{spec}_C
\end{tikzcd}$$

$$\begin{tikzcd}
& & \mathsf{shape}_A \arrow[dddd, "\mathsf{st}_A"'] & & \\
\mathsf{shape}_B \arrow[urr, "f_{\mathsf{shape}}"] \arrow[dddd, "\mathsf{st}_B"'] & & & & \mathsf{shape}_C \arrow[ull, "g_{\mathsf{shape}}"'] \arrow[dddd, "\mathsf{st}_C"] \\
& & & & \\
& \arrow[d, Rightarrow, "f_{\mathsf{mode}}"'] & & \arrow[d, Rightarrow, "g_{\mathsf{mode}}"] & \\
& & \mathsf{ext}_A^{op} & & \\
\mathsf{ext}_B^{op} \arrow[urr, "f^*_{\mathsf{spec}}"'] & & & & \mathsf{ext}_C^{op} \arrow[ull, "g^*_{\mathsf{spec}}"]
\end{tikzcd}$$

To compute the pushout, we first compute the corresponding pushout of specs and the pullback of shapes.

$$\begin{tikzcd}
& \mathsf{spec}_A \arrow[dl, "f_{\mathsf{spec}}"'] \arrow[drr, "g_{\mathsf{spec}}"] & & \\
\mathsf{spec}_B \arrow[drr, "s_{\mathsf{spec}}"'] & & & \mathsf{spec}_C \arrow[dl, "t_{\mathsf{spec}}"] \\
& & \mathsf{spec}_D &
\end{tikzcd}$$

$$\begin{tikzcd}
& \mathsf{shape}_A & & \\
\mathsf{shape}_B \arrow[ur, "f_{\mathsf{shape}}"] & & & \mathsf{shape}_C \arrow[ull, "g_{\mathsf{shape}}"'] \\
& & \mathsf{shape}_D \arrow[ull, "s_{\mathsf{shape}}"] \arrow[ur, "t_{\mathsf{shape}}"'] &
\end{tikzcd}$$

It is easy to see that $M : \mathsf{Spec}^{op} \longrightarrow \mathsf{Cat}$ maps the upper pushout to the pullback at the bottom of the induced cube.

$$\begin{tikzcd}
& & \mathsf{shape}_A \arrow[dddd, "\mathsf{st}_A"'] & & \\
\mathsf{shape}_B \arrow[urr, "f_{\mathsf{shape}}"] \arrow[dddd, "\mathsf{st}_B"'] & & & & \mathsf{shape}_C \arrow[ull, "g_{\mathsf{shape}}"'] \arrow[dddd, "\mathsf{st}_C"] \\
& & & \mathsf{shape}_D \arrow[ulll, "s_{\mathsf{shape}}"] \arrow[ur, "t_{\mathsf{shape}}"'] & \\
& \arrow[d, Rightarrow, "f_{\mathsf{mode}}"'] & & \arrow[d, Rightarrow, "g_{\mathsf{mode}}"] & \\
& & \mathsf{ext}_A^{op} & & \\
\mathsf{ext}_B^{op} \arrow[urr, "f^*_{\mathsf{spec}}"'] & & & & \mathsf{ext}_C^{op} \arrow[ull, "g^*_{\mathsf{spec}}"] \\
& & \mathsf{ext}_D^{op} \arrow[ull, "s^*_{\mathsf{spec}}"] \arrow[urr, "t^*_{\mathsf{spec}}"'] & &
\end{tikzcd}$$

If $f_{\mathsf{mode}}$ and $g_{\mathsf{mode}}$ were identities, i.e. if the two back faces of the cube were commutative, the fact that the bottom face is a pullback would induce a functor $\mathsf{st}_D : \mathsf{shape}_D \longrightarrow (\mathsf{spec}_D/\mathsf{Spec})^{op}$.

Since they are not, this functor must be constructed taking $f_{\mathsf{mode}}$ and $g_{\mathsf{mode}}$ into account. The image $\mathsf{st}_D(k)$ of a node $k$ of $\mathsf{shape}_D$ is now obtained as the unique arrow from the pushout at the top to the pushout at the bottom of the following cube.

$$\begin{tikzcd}
 & \mathsf{spec}_A \arrow[dl, "f_{\mathsf{spec}}"'] \arrow[drr, "g_{\mathsf{spec}}"] \arrow[ddd, "\mathsf{st}_A(\ell)"] & & \\
\mathsf{spec}_B \arrow[drr, "s_{\mathsf{spec}}"] \arrow[ddd, "\mathsf{st}_B(i)"'] & & & \mathsf{spec}_C \arrow[dl, "t_{\mathsf{spec}}"'] \arrow[ddd, "\mathsf{st}_C(j)"] \\
 & & \mathsf{spec}_D \arrow[ddd, dashed, "\mathsf{st}_D(k)"] & \\
 & \mathsf{mode}_A(\ell) \arrow[dl, "f_{\mathsf{mode}}(i)"'] \arrow[drr, "g_{\mathsf{mode}}(j)"] & & \\
\mathsf{mode}_B(i) \arrow[drr, "s_{\mathsf{mode}}(k)"'] & & & \mathsf{mode}_C(j) \arrow[dl, "t_{\mathsf{mode}}(k)"] \\
 & & \mathsf{mode}_D(k) &
\end{tikzcd}$$

Since $\mathsf{shape}_D$ is the pullback of $f_{\mathsf{shape}}$ and $g_{\mathsf{shape}}$, the node $k$ corresponds to a pair $\langle i, j\rangle$ of the nodes from $\mathsf{shape}_B$ and $\mathsf{shape}_C$, identified in $\mathsf{shape}_A$ as the node $\ell = f_{\mathsf{shape}}(i) = g_{\mathsf{shape}}(j)$. Of course, $i = s_{\mathsf{shape}}(k)$ and $j = t_{\mathsf{shape}}(k)$.

This construction gives the node part $\mathsf{mode}_D$ of $\mathsf{st}_D : \mathsf{shape}_D \longrightarrow \mathsf{ext}_D^{op}$, as well as the components of $s_{\mathsf{mode}}$ and $t_{\mathsf{mode}}$. The arrow part $\mathsf{step}_D$ is induced by the fact that the bottom of the cube is a pushout, using the naturality of $f_{\mathsf{mode}}$ and $g_{\mathsf{mode}}$. This also yields the naturality of $s_{\mathsf{mode}}$ and $t_{\mathsf{mode}}$. Finally we construct the guards for the edges of $\mathsf{shape}_D$. Given an edge $w : k \to k'$ of $\mathsf{shape}_D$ define

$$\begin{aligned}
\mathsf{guard}_D(w) \quad = \quad & s_{\mathsf{spec}}(\mathsf{guard}_B(s_{\mathsf{shape}}(w))) \\
& \wedge t_{\mathsf{spec}}(\mathsf{guard}_C(t_{\mathsf{shape}}(w)))
\end{aligned}$$

This completes the construction of the colimits in the lax comma category $D//M$. In order to complete the construction of the colimits in $\mathsf{ESpec}$, we need to supply the guards for the edges of $\mathsf{shape}_D$, and show that $s$ and $t$ satisfy the guard condition.

Given an edge $w : k \to k'$ of $\mathsf{shape}_D$ define

$$\mathsf{guard}_D(w) \quad = \quad s_{\mathsf{spec}}(\mathsf{guard}_B(s_{\mathsf{shape}}(w))) \wedge t_{\mathsf{spec}}(\mathsf{guard}_c(t_{\mathsf{shape}}(w)))$$

We need to prove that, for all $u : i \longrightarrow i'$ in $\mathsf{shape}_B$, all $v : j \longrightarrow j'$ in $\mathsf{shape}_C$, and all nodes $k$ in $\mathsf{shape}_D$, such that $s_{\mathsf{shape}}(k) = i$ and $t_{\mathsf{shape}}(k) = j$, holds

$$\begin{aligned}
\mathsf{mode}_D(k) \quad \vdash \quad & s_{\mathsf{spec}}(\mathsf{guard}_B(u)) \iff \bigvee_{\substack{s_{\mathsf{shape}}(w)=u \\ \mathsf{dom}(w)=i}} \mathsf{guard}_D(w) \\
\mathsf{mode}_D(k) \quad \vdash \quad & t_{\mathsf{spec}}(\mathsf{guard}_C(v)) \iff \bigvee_{\substack{t_{\mathsf{shape}}(w)=v \\ \mathsf{dom}(w)=j}} \mathsf{guard}_D(w)
\end{aligned}$$

Proving just one of these conditions will do, since they are symmetric; so let us take the first one. Unfolding the definition of $\mathsf{guard}_D$, it becomes

$$\mathsf{mode}_D(k) \quad \vdash \quad s_{\mathsf{spec}}(\mathsf{guard}_B(u)) \iff \bigvee_{\substack{s_{\mathsf{shape}}(w)=u \\ \mathsf{dom}(w)=k}} (s_{\mathsf{spec}}(\mathsf{guard}_B(u)) \wedge t_{\mathsf{spec}}(\mathsf{guard}_c(t_{\mathsf{shape}}(w)))$$

which is clearly equivalent to

$$\mathsf{mode}_D(k) \quad \vdash \quad s_{\mathsf{spec}}(\mathsf{guard}_B(u)) \Longrightarrow \bigvee_{\substack{s_{\mathsf{shape}}(w)=u \\ \mathsf{dom}(w)=k}} t_{\mathsf{spec}}(\mathsf{guard}_C(t_{\mathsf{shape}}(w))$$

This can be proved in three steps:

$$\mathsf{mode}_D(k) \quad \vdash \quad s_{\mathsf{spec}}(\mathsf{guard}_B(u)) \Longrightarrow s_{\mathsf{spec}} \circ f_{\mathsf{spec}}(\mathsf{guard}_A(f_{\mathsf{shape}}(u)) \tag{1}$$

$$\mathsf{mode}_D(k) \quad \vdash \quad s_{\mathsf{spec}}(\mathsf{guard}_B(u)) \Longrightarrow t_{\mathsf{spec}} \bigvee_{\substack{g_{\mathsf{shape}}(v)=f_{\mathsf{shape}}(u) \\ \mathsf{dom}(v)=j}} \mathsf{guard}_C(v) \tag{2}$$

$$\mathsf{mode}_D(k) \quad \vdash \quad s_{\mathsf{spec}}(\mathsf{guard}_B(u)) \Longrightarrow t_{\mathsf{spec}} \bigvee_{\substack{s_{\mathsf{shape}}(w)=u \\ \mathsf{dom}(w)=k}} \mathsf{guard}_C(t_{\mathsf{shape}}(w)) \tag{3}$$

(1) follows from the guard condition for $f$

$$\mathsf{mode}_B(i) \quad \vdash \quad f_{\mathsf{spec}}(\mathsf{guard}_A(f_{\mathsf{shape}}(u)) \iff \bigvee_{\substack{f_{\mathsf{shape}}(\widetilde{u})=f_{\mathsf{shape}}(u) \\ \mathsf{dom}(\widetilde{u})=i}} \mathsf{guard}_B(\widetilde{u})$$

and the logical fact that that $\mathsf{guard}_B(u)$ surely implies the disjunction on the right hand side, since it comes about as one of the disjuncts. Of course, the fact that $s_{\mathsf{spec}}$ preserves logical operations is also used.

(2) follows from (1), by replacing $s_{\mathsf{spec}} \circ f_{\mathsf{spec}}$ with $t_{\mathsf{spec}} \circ g_{\mathsf{spec}}$, which is equal, and then using the guard condition for $g$

$$\mathsf{mode}_C(j) \quad \vdash \quad g_{\mathsf{spec}}(\mathsf{guard}_A(f_{\mathsf{shape}}(u)) \iff \bigvee_{\substack{g_{\mathsf{shape}}(v)=f_{\mathsf{shape}}(u) \\ \mathsf{dom}(v)=j}} \mathsf{guard}_C(v)$$

Finally, (2) becomes (3) by noticing that $t_{\mathsf{shape}}$ induces the bijection between the index sets of the two disjunctions. Indeed, by the definition of $\mathsf{shape}_D$ as a pullback, the elements of

$$\{v \in \mathsf{shape}_C \mid g_{\mathsf{shape}}(v) = f_{\mathsf{shape}}(u) \wedge \mathsf{dom}(v) = j\}$$

are exactly the $t_{\mathsf{shape}}$-images of

$$\{w \in \mathsf{shape}_D \mid s_{\mathsf{shape}}(w) = u \wedge \mathsf{dom}(w) = k\}$$

(3) is, of course, just what we needed to prove, since $t_{\mathsf{spec}}$ surely preserves disjunction.

This completes the pushout construction.

**Explanation.** The pushout of specs is clear enough: the languages get joined together, and identified along the common part. The pullback of shapes produces the *parallel composition* of the behaviors they present. This is particularly easy to see for products, i.e. pullbacks over the final Espec. For example, a product of any $\mathsf{shape}$ with the two-node shape $\bullet \longrightarrow \bullet$ consists of the cylinder, with the two copies of $\mathsf{shape}$, and each two of their corresponding nodes connected by an edge. A product with the three-node shape $\bullet \longrightarrow \bullet \longrightarrow \bullet$ consists of three copies, similarly connected.

In general, the product of any two shapes $\mathsf{shape}_B$ and $\mathsf{shape}_C$ can be envisaged by putting a copy $S_n$ of $\mathsf{shape}_B$ at each node $n$ of $\mathsf{shape}_C$, and then expanding each edge $m \overset{u}{\to} n$ of $\mathsf{shape}_C$ into a cylinder from $S_m$ to $S_n$, i.e. a set of parallel edges, connecting the corresponding nodes. The initial node is the pair of the initial nodes of $\mathsf{shape}_B$ and $\mathsf{shape}_C$, whereas the final nodes are the pairs of final nodes.

In the resulting shape $\mathsf{shape}_B \times \mathsf{shape}_C$, each edge either comes from a copy of $\mathsf{shape}_B$ placed on a node of $\mathsf{shape}_C$, or from an edge of $\mathsf{shape}_C$ copied to connect two particular copies of a node of $\mathsf{shape}_B$; so it is either in the form $\langle\text{node of }\mathsf{shape}_C, \text{edge of }\mathsf{shape}_B\rangle$, or in the form $\langle\text{node of }\mathsf{shape}_B, \text{edge of }\mathsf{shape}_C\rangle$. A moment of thought shows that each path through $\mathsf{shape}_B \times \mathsf{shape}_C$ corresponds to a shuffle of a path through $\mathsf{shape}_B$, and a path through $\mathsf{shape}_C$; and that every such path comes about as a unique path in $\mathsf{shape}_B \times \mathsf{shape}_C$. In this sense, $\mathsf{shape}_B \times \mathsf{shape}_C$ is the *parallel composition* of $\mathsf{shape}_B$ and $\mathsf{shape}_C$.

A pullback extracts a part of such product, identified by a pair of shape morphisms $\mathsf{shape}_B \longrightarrow \mathsf{shape}_A \longleftarrow \mathsf{shape}_C$. Since the initial node must be preserved, the initial node of the product will surely be contained in the pullback. The set of final nodes may be empty in general.

For each pair of nodes $\langle i, j\rangle$, contained in the pullback $\mathsf{shape}_D$ as the node $k$, the corresponding state description is constructed as the pushout $\mathsf{mode}_D(k)$ of $\mathsf{mode}_B(i)$ and $\mathsf{mode}_C(j)$ on the above diagram. As a theory, this state description may be inconsistent. Indeed, if $B$ and $C$ are not independent, but have a shared part $A$, their parallel composition may be globally inconsistent, in the sense that $\mathsf{spec}_D$ may be inconsistent; or some of the pairs of states that may come about in shuffling their computation paths may be inconsistent, which makes such paths computationally impossible. Inference tools can be used to eliminate inconsistent/unreachable modes from the colimit Espec.

Despite the seeming complexity and mathematical depth in the description of the colimit, the actual computation is relatively simple. There are just three steps:

- pullback of shapes;
- pushout of specs; (the guards can be directly computed at this point)

- the pushout extensions of modes and steps.

The first two steps are simple and well known. The third one amounts to computing a pushout of theories for each mode, and using the universality of each such pushout to generate the steps from it – Epoxi's colimit algorithm returns both the cocone and a generator of interpretations that witness the universality of the apex.

### 3.8 Parameterization: Modeling the Environment

In system design, the need arises to specify the properties and behavior of a component's environment, including required behaviors, invariant properties, and required services. The correctness of the component's behavior follows from the assumption that the environment behaves as specified, together with the internal structure and behavior of the component. The concept of parameterized Especs neatly satisfies this need.

A parameter to an Espec is an Espec that models the environment – what behavior and properties the component expects of the environment, and what services it requires. The binding of a parameter to the environment is given by an Espec morphism – the environment is expected to be a refinement of the parameter. The environment will typically have much more structure and behavior than is specified by the parameter, but it must have at least as much as is required for the correct operation of the component.

A parameterized Espec is presented as an import morphism from the parameter to the body Espec with special properties. Intuitively, the semantics of a parameter requires that the behavior of the component consistently extends the behavior of environment – no behaviors of the environment allowed by the parameter can be ruled out by restrictions in the body of the component. In [13], parameterized higher-order logic specifications (as in the Specware system) are treated as coherent functors. Parameters are refined contravariantly (cf [23]).

Two other properties of parameterized specs are important ([23, 13] and there is good reason to believe that analogous properties hold for Especs, although we have not yet proved them.

1. Parameterized specs are closed under composition
2. Parameterized specs are stable under pushout – that is, if the pushout of $A \rightarrow B \leftarrow C$ is $D$ and $A \rightarrow B$ is a parameterized spec, then $C \leftarrow D$ is also a parameterized spec.

In the body of a parameterized Espec (a *p-Espec*), the prog may include both offered and required events and procedure calls. To distinguish these, we label offered events and procedures as *input* items because the component promises to accept any external request for them. For example, an offered event may correspond to a message being sent to the component; i.e. an input from the environment. Dually, we label required events and procedures as *output* items because the environment of the component is required to accept any internally-generated request for them. For example, a required event may correspond to a message being sent to the environment; i.e. an output from the component to the environment. Required/output events

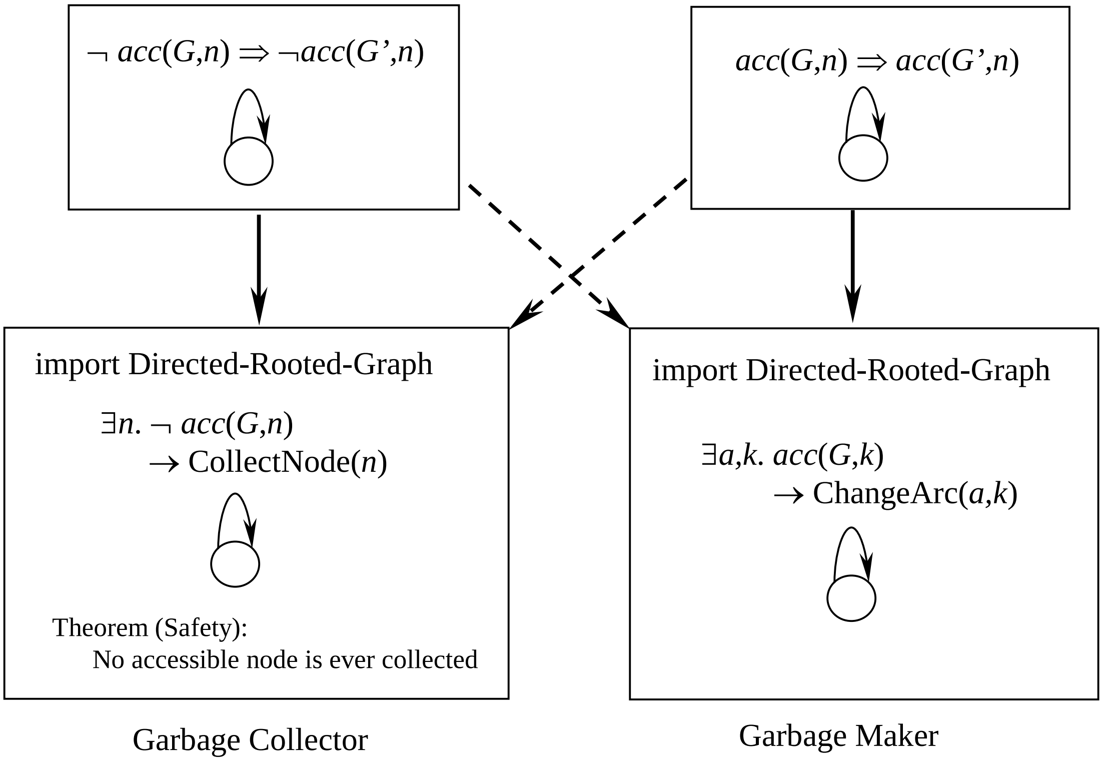


Figure 2: Composition of a Garbage Maker and a Garbage Collector

and procedures are introduced in the parameter of a p-Espec, and offered/input events and procedures are introduced in the body of a p-Espec. The binding morphism from a parameter to the environment must be *polarity-changing* in the sense that output items must map to input items in the environment, and vice-versa. In contrast, an ordinary morphism between Especs must be *polarity-preserving.*

Figure 2 shows the composition of an abstract garbage maker and a garbage collector [14]. Note that both components have required behaviors of the environment that are presented in their parameter. These behaviors are invariants that each component requires the other to preserve in its own actions. The proper functioning of each component depends on the other fulfilling its obligation to satisfy the corresponding invariant. In particular, the garbage collector expects that the garbage maker preserves the property of inaccessibility ($acc(G, n)$ means that node $n$ is accessible in directed graph $G$ which represents the nodes and pointers in memory). Conversely, the garbage maker expects that the garbage collector preserves the accessibility of nodes. The lesson here is that environmental requirements can be more than just availability of procedures. We will see in Section 4 that the formalism also supports specification of constraints on the sequencing of actions of several components.

### 3.9 Timed Especs

Several of our examples required the composition and coordination of timing constraints. We found that the simplest approach was to add clocks as primitive concepts to Especs [2]. Variables can have type `Clock` which means that their value increases continuously with derivative one relative to real time. Clock variables can be assigned to, so `c := 0` resets clock `c` to 0. The predicates `=` and `<=` are used to compare clocks and time constants (usually integers in this report). Clock variables and constraints refine and compose just like other variables and constraints in the category of Especs.

### 3.10 Cleaning-up after Composition

The outcome of composing a system via colimit is a new system specification, a parameterized Espec, that specifies the required behaviors of its environment as well as system (internal) properties, behaviors, and services. A collection of components are composable if there is some environment that satisfies the parameter specification; i.e. if the resulting parameter is consistent and the body is consistent.

When we compute the colimit, of an architecture, we take a kind of Cartesian product of the possible behaviors of each component and connector. Not all combinations of these behaviors are compatible. For example, two simultaneous but different assignments to the same variable lead to an inconsistent state. More generally, the colimit may contain modes that are inconsistent, and therefore unreachable. Therefore, after composition (i.e. computing the colimit), it is necessary to analyze the composition for error states and to eliminate them. The result is a refinement of the composition that has fewer (or no) behaviors leading to blocked states (nonfinal states from which there is no exit). Typical examples of error states are:

- *unreachable modes* – including inconsistent modes
- *no enabled transitions in a given state*
- *an enabled transition causes an error* – e.g. divide by zero
- *refusal of one component to accept an input from another*
- *uncoordinated timing conditions*

The avionic mission control system in Section 4.2 works through the clean-up process for the composition of timed systems. Essentially, generic timing constraints on modes and transitions are used to set up a definite constraint system that can be solved in linear time. The result is to tighten temporal guards in order to preclude joint behaviors that lead to error states (typically that one component sends a message that another is not yet prepared to receive).

Each type of error state can be eliminated by special means. In this project we performed preliminary investigation of (1) elimination of inconsistent modes by means of theorem-proving to detect inconsistency, and (2) timing dysfunctions that are handled by constraint propagation. Ongoing projects at Kestrel are addressing the other kinds of error states.

### 3.11 Especs and Software Architectures

In this section we compare and contrast the Espec formalism with the classical concept of software architecture.

The Espec formalism provides fresh insight into the classical concept of software architecture, which is based on the distinction between components and connectors. A system is modeled as a collection of components that are interconnected via connectors that mediate the interaction between components. While the distinction of component and connector may be useful conceptually, it is not fundamental. Especs can naturally represent the behavior both of components and connectors. In the following we use the term component to denote both components and connectors.

As emphasized earlier, there is a fundamental and pervasive distinction between a component or system, and its environment [1]. With respect to specifying a component or connector, this manifests itself in distinguishing two kinds of services: its *offered* and *required* services. A component offers certain services to its environment and these offered services can be encapsulated by an Espec that refines to the component itself. This provided-service-Espec is usually called the *interface*, and the component is an implementation (i.e. a refinement) of it. Dually, a component may require certain services of its environment. This requirement can be encapsulated by a parameter Espec (See Section 3.8).

The offered and required service Especs of a component come with a warranty: if the environment provides services that satisfy a component's required services, then the component guarantees to provide its offered services.

The service specs provide a way to decompose the verification of a system into (i) correctness of each component (i.e. establishing its warranty), and (ii) interconnecting the components via a diagram of Especs.

The distinction of service specs also gives rise to several composition possibilities via the kind of interconnection:

- *Pushout of a parametric Espec with an offered Espec* – To show that a component can be correctly composed into a system, we must provide a binding morphism from its requirement/parameter Espec to the offered service Espec of its environment, which shows how the component's environment satisfies the requirement. The pushout operation generates the component instantiated with the particular services provided by the environment. The required and offered services are identified in the colimit.
- *Pushout of offered Especs* – Using a pushout to compose the offered Especs (aka interfaces) of two components has the effect of generating the offered spec/interface of the composed components.

In terms of software architecture, the *ports* of a component (respectively the *roles* of a connector) seem to refer both to required and provided services without distinction. Yet the Espec formalism reveals subtle semantic distinctions between them - a parameterized Espec is morphism with special properties [13] (essentially the body cannot introduce new properties of the environment and it cannot restrict the behavior of the environment) whereas the morphism from the offered-services-Espec is a general refinement morphism. Another distinction is revealed by refinement – when a component specification is refined, it's input items (offered services) may increase, but its output items (required services) may only decrease. This property is call contravariance. Intuitively it can be understood as follows: when a system is specified to exhibit property P in environments of type E, then an implementation must satisfy P and perhaps other properties (i.e. strengthening P), *and* it must operate in at least the environments satisfying E and perhaps others (i.e. weakening E). So the environment model weakens under refinement and the system specification strengthens.

Consequently the Espec formalism provides fresh insight into a necessary elaboration or refinement of software architecture theory. This insight stems from our concerns about (1) precise semantics and correctness of composition, and (2) the polarity between system and environment. Furthermore, since Especs are formulated with refinement in mind, it is natural to compose a system at very abstract levels and conceive of refining both the connectors and the components. This helps improve the clarity of system at the highest level, and to help a reader understand the details of the final system structure by introducing information in a staged manner, via refinements.

# 4 Examples

To illustrate the abstract concepts in the previous section, we present several examples in detail here.

1. *Distributed Sensor Network* – The first example has been demonstrated at project workshops and shows the composition of a dynamic architecture for a distributed sensor network. Epoxi automatically translates the Espec architecture to running code on the UC Berkeley motes [9]. The sensor network dynamically adapts itself in response to faults in its underlying communication grid (short-distance radio links).

2. *Avionic Mission Controller* – The second example focuses on system specification and design-time composability/compliance checking. The problem is to specify a avionic mission controller together with its requirements on external components that provide asynchronous communication with a radar unit. The problem places particular emphasis on specifying end-to-end timing requirements in the architecture and enforcing compliance at design-time.

3. *Distributed Garbage Collection* – A third example, which can be found in [14], illustrates the use of parameters to specify behavioral and logical invariants on the environment. In this case there are two components interacting on a shared memory - a garbage maker and a garbage collector. Each requires certain behavioral properties of the other in order that each can provide its service. This provides a concise example of how to precisely specify and compose multi-agent systems that both compete and cooperate with each other.

4. *Plant Monitoring System* – A fourth example, which can be found in [21], explores the elaboration of Especs with syntax and semantics for specifying the physical world. The problem is to monitor a steel milling operation and to provide a warning within a given time bound when conditions fall outside a safe region. The issue is specifying the physical world activities together with the sensors that measure physical world quantities and output bits that can be treated computationally, and to model the inherent uncertainty of such measurements and the ensuing effects of that uncertainty upon computations.

Each of these examples treat a different mix of issues related to system specification and composability issues. The Espec formalism provides the framework for addressing the question of what happens when the system cannot be proven at design-time to be composable; e.g. some properties are only knowable at run-time. This case is revealed by failure to verify the morphisms that interconnect components in the architecture. Knowing which properties fail to hold allows us to generate probes and gauges that verify them at run-time, signaling the need for special handling when they do not hold.

## 4.1 Sensor Nodes

### 4.1.1 Problem Description

We are given a set of low-cost, low-power autonomous hardware devices (called "Motes"), which are equipped with sensors and are able to communicate with each other via radio signals [9]. They may be scattered in the field in order to track movements.

The basic system architecture comprises sensor components and radio connectors from each sensor to one of several data fusion components. When we attempt to refine the basic architecture to the motes, we find that the radio link has low reliability. To cope with such a fail-silent component/connector, a standard approach is to impose a heartbeat probe and a clock-based gauge that decides when the component/connector has failed. We formalized this heartbeat gauge architecture as an Espec diagram and formally composed it with the basic architecture to produce a dynamic adaptive sensor-net system. The Espec compiler translates the system to C and then into machine code for motes. The running system dynamically forms a network to flow sense data to the data fusion components, and dynamically adapts the network when a connector is gauged to have failed.

The Especs presented here specify the code that is running on each of these motes, including an abstraction of their tracking, communication, and self-organization behaviors. The model assumes that the environment of the mote transmits a stream of events which are interpreted and processed by the mote, e.g. receive message (over a radio channel), clock tick, initialize. While changing from one state to the other, the Espec is able to perform updates of the locally defined variables as well as to invoke commands, e.g. send message, clock initialization.

The example will be presented in three steps: first the Espec for specifying the target tracking functionality of the application is given. In this model, we have abstracted from the fact that the communication links between the sensor node are unstable, because of the unstable nature of radio signals. Next, an Espec covering the detection and repair of broken communication links between sensor nodes is given. That is, we present a domain-independent abstract Espec for a heartbeat probe and a clock-based gauge that decides when the component/connector has failed. This heartbeat gauge architecture is then formally composed with the basic architecture to produce a dynamic adaptive sensor-net system. The Espec compiler translates the system to C and then into machine code for motes. The running system dynamically forms a network to flow sense data to the data fusion components, and dynamically adapts the network when a connector is gauged to have failed.

### 4.1.2 Espec: Target Tracking

The following Espec specifies the simple target tracking state machine running on each sensor node. Whenever the environment (i.e. the underlying operating system) transmits an event, the parameter `cmd` is set to the value of the specific event. In this simplified case, the detection of a radio message on channel 12 is regarded as a signal from the target. The task of a sensor node is to count number of target signals during the time between two clock ticks and send the result to the data collector node (abbreviated as Daco throughout the example). The data collector is another node that is also equipped with radio communication facilities.

The initial state is Initialize which automatically transitions to the Receive mode and initializes various variables. In Receive mode, the mote waits for an event/command which causes a transition to Dispatch. In Dispatch mode, the value of cmd is checked. If the operating system is in its initialization phase, it will issue an Init event, causing the Espec to reinitialize its local variables. In case of an ClockEvent (a clock tick) the state transition causes the sensor node to reset its targetCnt after it has been sent to the data collector node.

This graphical representation of the above state machine can be transformed to a textual version as follows:

```
ReceiveCommands
 = proc  % models the environment sending a stream of events/cmds
    import Common
    var cmd:Event
    A = initial mode
          step -> A is !cmd end-step
          end-mode
   end-proc
```

```
SensorMote
 = proc  % obtain Common from the parameter
    parameters ReceiveCommands

    var CurrentDaco : NodeId
    var targetCnt : Nat
    sendToDaco = proc parameters {CurrentDaco : NodeId,
                                  targetCnt : Nat}
                 end-proc

   Initialize = initial mode
                 step -> Receive is
                   clockInit(Send, 2), % clock to xmit 2 ticks/sec to ch Send
                   targetCnt := 0,
                   CurrentDaco := 0   % should acquire this value
                  end-step
                 end-mode

   Receive = mode
       GetCmd = step -> Dispatch is ?cmd  end-step
       end-mode

   Dispatch = mode
     ResetStep = step -> Receive is
       if cmd matches Init
       then targetCnt := 0
       end-step
    DetectTarget = step -> Receive is
       if  cmd matches ReceiveMsgCh12 _  % ch12?msg
       then targetCnt := targetCnt + 1
       end-step
    SendSensorData = step -> Receive is
       if cmd matches ClockEvent(Send)
       then  sendToDaco(CurrentDaco,targetCnt),
             targetCnt := 0
       end-step
   end-mode
 end-proc
```

As mentioned earlier, this state machine doesn't take into account the unstable nature of radio signals, which are used as communication means between the sensor nodes and the data collector node(s).

The following Espec specifies the algorithm for detecting and reacting to failures of the communication links between nodes. The principle of this algorithm is that the nodes listen to "heart beats" sent out from their neighbor nodes. If for a certain amount of time the heart beat of a neighbor has not been detected, then the link to this node is cut off. The state machine uses the nodes `initialize`, `Receive`, and `Dispatch` as the previous did. We want to point out, that the model given below is independent from the target tracking model:

```
HeartBeatAdaption
 = proc
    parameters ReceiveCommands
    var msg : string

    var GotHeartBeat : Boolean
    var CurrentDaco : NodeId
    var AlternateDaco : NodeId
    var HaveAlternateLink : Boolean

   Initialize = initial mode
                 step -> Receive is
                   clockInit(HB,1),  % clock to xmit 2 ticks/sec to ch HB
                   HaveAlternateLink := false
                  end-step
                 end-mode

   Receive = mode
       GetCmd = step -> Dispatch is ?cmd  end-step
       end-mode

   Dispatch = mode
      ProcessHeartBeat: step -> Receive is
        if cmd matches ReceiveMsgCh8 msg
        then GotHeartBeat := true
        end-step
     Reconnect = step -> Receive is
       if (cmd matches ClockEvent(HB)) & ~GotHeartBeat & HaveAlternateLink
       then CurrentDaco := AlternateDaco,
            GotHeartBeat := false,
            HaveAlternateLink := false
       end-step
     UpdateAlternate = step -> Receive is
       if cmd matches ReceiveMsgCh7(msg)
       then AlternateDaco := msg,
            HaveAlternateLink := true
       end-step
   end-mode
  end-proc
```

### 4.1.3 Composed System

The effect of composing the SensorMote model and the FaultAdapation model is to create a SensorMote model augmented with FaultAdaptation with higher reliability due to the dynamic adaptation of the communication link.

```
RC
 = proc  % models the environment sending a stream of events/cmds
    import Common
    var cmd:Event
    A = initial mode
          step -> A is !cmd end-step
          end-mode
   end-proc

Abs
 = proc
    parameters RC
    var CurrentDaco : NodeId
    Initialize = initial mode
                  step -> Receive is end-step
                 end-mode

    Receive = mode
       GetCmd = step -> Dispatch is ?cmd  end-step
       end-mode

    Dispatch = mode
     DR1 = step -> Receive is end-step
     DR2 = step -> Receive is end-step
     DR3 = step -> Receive is end-step
     DR4 = step -> Receive is end-step
     DR5 = step -> Receive is end-step
     DR6 = step -> Receive is end-step
    end-mode
 end-proc
```

```
ABStoSensorMote
         = morphism ABS -> SensorMote
            parameter  % the parameter refines contravariantly!
                       morphism ReceiveCommands -> RC
                        parameter
                        commonspec {} % Common -> Common
                        structure {A +-> A}
            commonspec % the commonspec is covariant
                       {CurrentDaco +-> CurrentDaco}
            structure  % the natXform component, expressed contravariantly
                       {Initialize +-> Initialize,
                        Receive    +-> Receive,
                        Dispatch   +-> Dispatch,
                        ResetStep       +-> DR1,
                        DetectTarget    +-> DR2,
                        SendSensorData  +-> DR3},

ABStoHeartBeatAdaption
         = morphism ABS -> HeartBeatAdaption
            parameter  % the parameter refines contravariantly!
                       morphism ReceiveCommands -> RC
                        parameter
                        commonspec {} % Common -> Common
                        structure {A +-> A}
            commonspec % the commonspec is covariant
                       {CurrentDaco +-> CurrentDaco}
            structure  % the natXform component, expressed contravariantly
                       {Initialize +-> Initialize,
                        Receive    +-> Receive,
                        Dispatch   +-> Dispatch,
                        ProcessHeartBeat +-> DR4,
                        Reconnect        +-> DR5,
                        UpdateAlternate  +-> DR6}
```

Here is the computed pushout, the composition of SensorMote and HeartBeatAdaption that results in a new module that boths acts as a sensor and dynamically adapts its radio communication link.

```
ReceiveCommands
 = proc  % models the environment sending a stream of events/cmds
    import Common
    var cmd:Event
    A = initial mode
          step -> A is !cmd end-step
          end-mode
   end-proc
```

```
FaultAdaptiveSensorMote
 = proc  % obtain Common from the parameter
    parameters ReceiveCommands

    var CurrentDaco : NodeId
    var targetCnt : Nat
    var msg : string

    var GotHeartBeat : Boolean
    var CurrentDaco : NodeId
    var AlternateDaco : NodeId
    var HaveAlternateLink : Boolean

    sendToDaco = proc
                  parameters {CurrentDaco : NodeId,
                              targetCnt : Nat}
                 end-proc

   Initialize = initial mode
                 step -> Receive is
                   clockInit(Send, 2), % clock to xmit 2 ticks/sec to ch Send
                   targetCnt := 0,
                   CurrentDaco := 0,   % should acquire this value
                   clockInit(HB,1),  % clock to xmit 2 ticks/sec to ch HB
                   HaveAlternateLink := false
                  end-step
                 end-mode

   Receive = mode
       GetCmd = step -> Dispatch is ?cmd  end-step
       end-mode

   Dispatch = mode
      ResetStep = step -> Receive is
       if cmd matches Init
          then
          targetCnt := 0
          end-step

     DetectTarget = step -> Receive is
       if  cmd matches ReceiveMsgCh12 _  % ch12?msg
       then targetCnt := targetCnt + 1
       end-step

     SendSensorData = step -> Receive is
       if cmd matches ClockEvent(Send)
       then  sendToDaco(CurrentDaco,targetCnt),
             targetCnt := 0
       end-step

     ProcessHeartBeat: step -> Receive is
        if cmd matches ReceiveMsgCh8 msg
        then GotHeartBeat := true
        end-step

     Reconnect = step -> Receive is
       if (cmd matches ClockEvent(HB)) & ~GotHeartBeat & HaveAlternateLink
       then CurrentDaco := AlternateDaco,
            GotHeartBeat := false,
            HaveAlternateLink := false
       end-step

     UpdateAlternate = step -> Receive is
       if cmd matches ReceiveMsgCh7(msg)
       then AlternateDaco := msg,
            HaveAlternateLink := true
       end-step
   end-mode

 end-proc
```

## 4.2 Mission Controller

This example illustrates the architectural composition of two components (a mission controller and a radar unit) via a connector (effectively a Remote Procedure Call mechanism). We show how the classical concepts of architectures as components and connectors is naturally expressed in the Espec formalism, and moreover, provides a semantically precise foundation for gauging logical properties, timing properties, and behavioral constraints. This example also shows how the synthesis of glue code enables a dramatically broader concept of "compliance" at composition-time. The example highlights the use of post-composition propagation to coordinate the timing behavior of the system. This propagation is a key part of assessing composability of the system - if timing guards refine to false, then the system works in no environment.

This section shows how Especs support an architectural approach to system design. Components can be formally presented as Especs with two kinds of interfaces: (1) *required services* are specified by parameter Especs that model the structure, behavior, and services that the component expects of its context/environment, and (2) *offered services* are specified by Especs that model the services provided by the component. The correctness of an open-systems component is established by proving that the component will provide its offered services assuming that its environment supplies the required services specified in the parameter. The protocol of interaction between a component and its environment is specified by the component's state machine, which constrains the interleaving of input, internal, and output actions.

The architecture of a system is presented as a diagram of Especs in which *required services* are bound to offered services by means of Espec morphisms, and the offered services of subsystems are composed by colimit of subdiagrams.

In the following scenario, a radar unit component and a mission controller component are composed via a generic connector that models a synchronous communication channel. The Epoxi specifications for the components and connector model the behavior, structure, and roles/ports for each. They are connected together by means of a diagram and composed via a colimit of Especs.

The following specs are abridged in order to save space and focus on essentials, in particular, the architectural structure and the correct composition with respect to timing constraints and data-level interoperation.

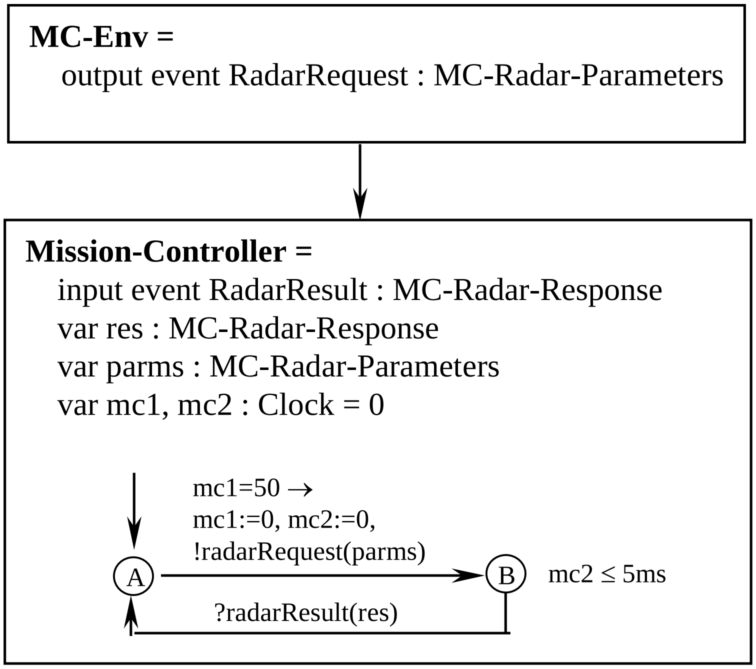

MC-Env =
output event RadarRequest : MC-Radar-Parameters
Mission-Controller =
input event RadarResult : MC-Radar-Response
var res : MC-Radar-Response
var parms : MC-Radar-Parameters
var mc1, mc2 : Clock = 0
mc1=50 →
mc1:=0, mc2:=0,
!radarRequest(parms)
A
B
mc2 ≤ 5ms
?radarResult(res)

Figure 3: **Mission Controller**

### 4.2.1 Components

The main component is the mission controller. To illustrate basic features of composability, we simply present an abstraction of the controller that interacts with the radar unit. More details may be found in [16]. Here, the parameter Espec models the expectations that `MISSION-CONTROLLER` has of its environment. In particular, it assumes

1. ability to handle a request for a radar image every 50ms (20 requests per second),
2. at most a 5ms response time between each request and the corresponding response.

A textual presentation is given below and a simplified depiction is shown in Figure 3. The notation `!m` indicates the transmission of an event `m` and `?m` indicates the reception of an event `m`. Note that the program of `Mission-Controller` specifies the interaction of the controller with its environment via the sequencing and timing of events. Actual mission processing activities are abstracted away in mode `A` – modes denote activities, so `A` can refine to a submachine carrying out extensive processing.

```
MC-parameter = espec
    sort MC-Radar-Parameters
            =  { gain                : Real,
                 dwell               : Real Microsecond,
                 frequency           : Real Hertz,
                 emission-direction  : Radian }
    output event RadarRequest : MC-Radar-Parameters
end-espec

Mission-Controller = espec
    import MC-parameter
    sort MC-Radar-Response
                = { time-stamp       : Real Microsecond,
                    detect-object?   : Boolean,
                    object-location  : Lat-Long}
    input event RadarResult : MC-Radar-Response

    var res : MC-Radar-Response
    var parms : MC-Radar-Parameters
    var mc1, mc2 : Clock

 A = mode
      axiom mc2 <= 5ms
      AB: step -> B
        mc1 = 50ms ->
          mc1 := 0, mc2 := 0,
          !RadarRequest(parms)
        end-step
     end-mode

 B = mode
      BA: step -> A
        ?RadarResponse(res)
        end-step
     end-mode
end-espec
```

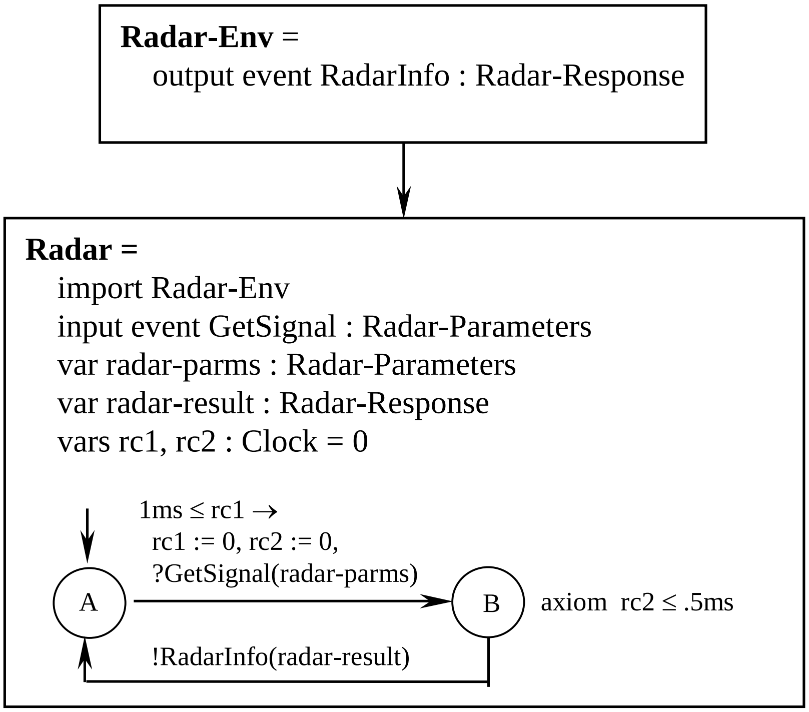

Radar-Env =
output event RadarInfo : Radar-Response
Radar =
import Radar-Env
input event GetSignal : Radar-Parameters
var radar-parms : Radar-Parameters
var radar-result : Radar-Response
vars rc1, rc2 : Clock = 0
1ms ≤ rc1 →
rc1 := 0, rc2 := 0,
?GetSignal(radar-parms)
A
B
axiom rc2 ≤ .5ms
!RadarInfo(radar-result)

Figure 4: **Radar Unit**

A radar component accepts requests for radar sweeps and provides the processed responses. Our specification abstracts away the actual model of radar emission as well as the processing of the return signal. The radar requires that the environment makes requests with a time separation of no less than 1ms (i.e. no faster than 1000Hz). In turn, it offers a .5ms response time to each `GetSignal` request.

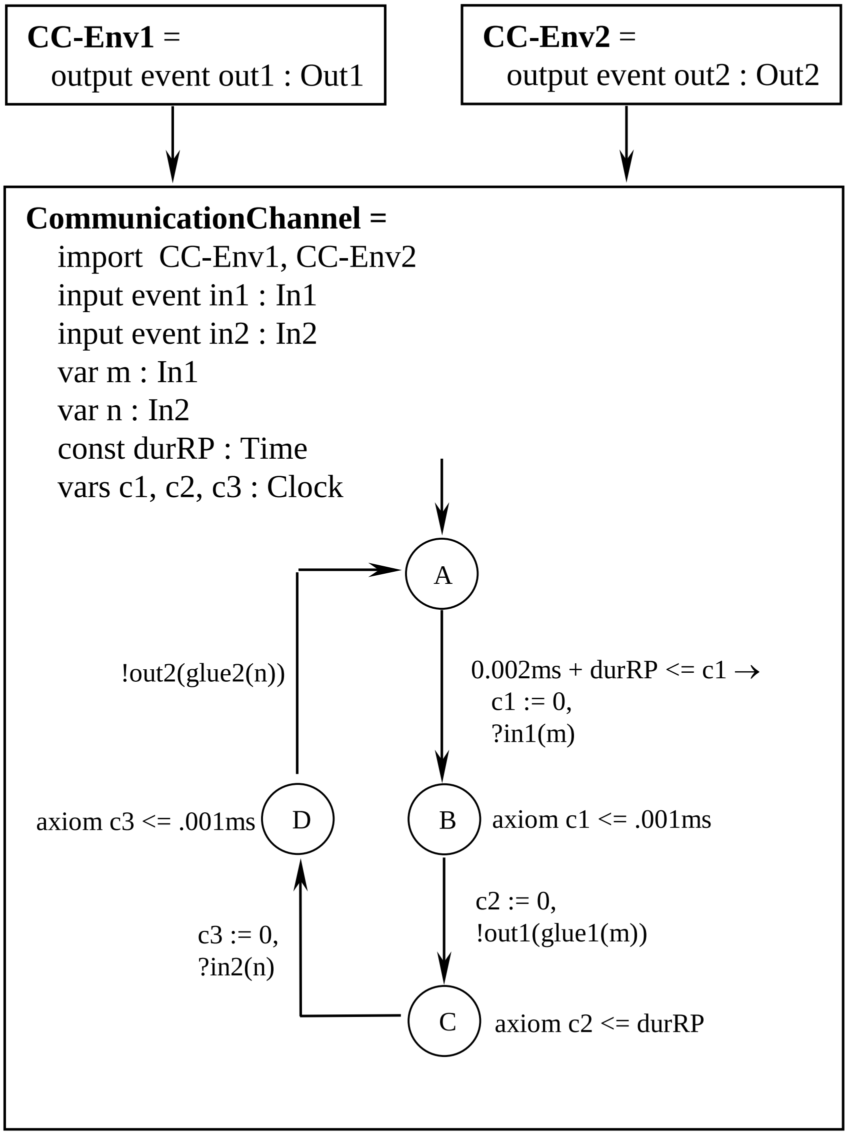
```
RADAR-Env = espec
    sort Radar-Response
                = { time-stamp       : Real Millisecond,
                    detect-object?   : Boolean,
                    object-location  : Range-Bearing}
    output event RadarInfo : Radar-Response
end-spec

RADAR = espec
    import RADAR-Env
    sort Radar-Parameters
            =  { gain                : Real,
                 frequency           : Real Hertz,
                 dwell               : Real Microsecond,
                 emission-direction  : Radian}
    input event GetSignal : Radar-Parameters
    var radar-parms : Radar-Parameters
    var radar-result : Radar-Response
    vars rc1, rc2 : Clock = 0

 A = initial mode
      AB: step -> B
        1ms <= rc1 -> rc1 := 0, rc2 := 0, ?GetSignal(radar-parms)
        end-step
     end-mode

 B = mode
      axiom rc2 <= .5ms
      BA: step -> A
        !RadarInfo(radar-result)
        end-step
     end-mode

end-espec
```

### 4.2.2 Connector

In the scenario, we decide to compose the mission controller with a radar unit by means of a connector that provides synchronized communication service with glue (data translation) both in and out. Transmission time is specified to be at most 1ms in both directions. The connector is specified in terms of generic types that will be refined in context. The translation functions `glue1` and `glue2` are only constrained by the axiom `glue`. A glue-code generator is needed to refine this spec to executable code. In many cases the data translators will be the identity functions, in which case they can be optimized away by further transformation of the code.

Note that the transition diagram in the body of `Communication-Channel` specifies both the protocol of interaction between the components that it connects, both the sequencing of shared events and various timing properties. In words, `Communication-Channel` behaves as follows: it receives a message `in1` and then translates it and transmits it out as `out1` no more than .001ms later. No longer than `durRP` milliseconds later it expects to receive a message `in2` which it translates and sends as `out2` no more than .001ms later. This pattern repeats indefinitely. The channel itself is relatively fast - it can accommodate requests at rates up to 500KHz. When we compose it to other components that operate at slower rates, there will be a problem of compliance - can the respective rates be coordinated to achieve the desired behavior of the whole system?

Note also that the `Communication-Channel` has two parameters. It is natural to separate the requirements on the two components that the connector mediates.



```
CC-Env1 =
  output event out1 : Out1

CC-Env2 =
  output event out2 : Out2

CommunicationChannel =
   import  CC-Env1, CC-Env2
   input event in1 : In1
   input event in2 : In2
   var m : In1
   var n : In2
   const durRP : Time
   vars c1, c2, c3 : Clock
```

Figure 5: **Communication Channel**

```
CC-Env-1 = espec
    sort Out1
    output event out1 : Out2
end-espec

CC-Env-2 = espec
    sort Out2
    output event out2 : Out2
end-espec

Communication-Channel = espec
  import CC-Env-1, CC-Env-2
  sorts In1, in2
  input event in1 : In1
  input event in2 : In2
  var m : In1
  var n : In2
  const durRP : Time

  op P : In1 * Out2 -> Boolean
  op Q : Out1 * In2 -> Boolean
  op glue1 : In1 -> Out1
  op glue2 : In2 -> Out2
  axiom glue is
    fa(in1:In1, out1:Out1,
       in2:In2, out2:Out2)
      (out1 = glue1(in1)
       & Q(out1,in2)
       & out2 = glue2(in2)
       => P(in1,out2))

  vars c1, c2, c3 : Clock

 A = initial mode  % input message m
      AB: step -> B
         0.002ms + durRP <= c1 -> c1 := 0, ?in1(m)
        end-step
     end-mode

 B = initial mode  % output the translation of m
      axiom c1 <= .001ms
      BC: step -> C
         c2 := 0, !out1(glue1(m))
        end-step
     end-mode

 C = initial mode  %wait for response n
```

```
      axiom c2 <= durRP
      CD: step -> D
         c3 := 0, ?in2(n)
        end-step
     end-mode

 D = initial mode  % output the translation of n
      axiom c3 <= .001ms
      DA: step -> A
         !out2(glue2(n))
        end-step
     end-mode

end-espec
```

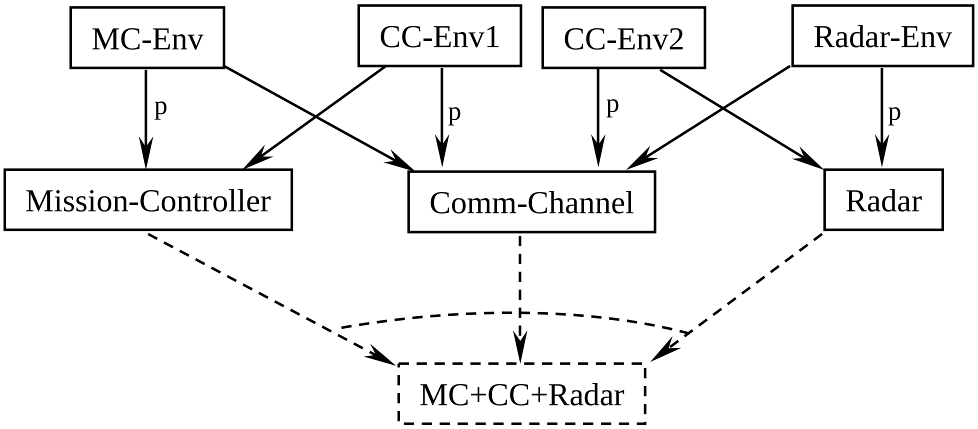


Figure 6: **Composing the Components and Connector**

### 4.2.3 Composing the Architecture

Figure 6 shows how we interconnect the components and connector of the mission control system in Epoxi. The new interconnection information is represented in the morphisms that bind parameter/environment Especs to bodies of other components. In particular, since the channel provides the environment for the Mission controller and the Radar unit, we need to develop morphisms from `Radar-Env` and `MC-Env` to `Communication-Channel`. We do not discuss techniques for automating the construction of these morphisms ( see [19]), but the construction is relatively straightforward here.

The resulting colimit is the instantiated Mission-Controller operating concurrently with the communication channel and the radar.

```
MCEnvToCC
     = morphism : MC-Env -> Communication-Channel
         commonspec { MC-Radar-Parameters +-> In1 }
         structure  { in1                 +-> RadarRequest }
```

```
CCEnv2toMC
     = morphism : CC-Env2 -> MC
         commonspec { Out2        +-> MC-Radar-Parameters }
         structure  { RadarResult +-> out2}
```

```
RadarEnvToCC
     = morphism : Radar-Env -> Communication-Channel
         commonspec { Radar-Response +-> In2 }
         structure  { in2            +-> GetSignal}
```

```
CCEnv2toRadar
     = morphism : CC-Env2 -> Radar
         commonspec { Out1      +-> Radar-Parameters }
         structure  { RadarInfo +-> out1}
```

The generated Espec for the colimit (MC-CC-Radar) is shown in Figure 7.

Note that the types and interface to the Radar have been concretized, but the glue code is specified but not defined.

Note also that input/output events that are identified lose their polarity and become internal events (e.g. RadarResult).

```
Mission-Control-System = espec
  sort MC-Radar-Response
                = { time-stamp       : Real Microsecond,
                    detect-object?   : Boolean,
                    object-location  : Lat-Long}
  event RadarResult : MC-Radar-Response
  sort MC-Radar-Parameters
            =  { gain                : Real,
                 dwell               : Real Microsecond,
                 frequency           : Real Hertz,
                 emission-direction  : Radian }
  event RadarRequest : MC-Radar-Parameters
  var parms : MC-Radar-Parameters
  var mc1, mc2 : Clock

  sort Radar-Parameters
            =  { gain                : Real,
                 frequency           : Real Hertz,
                 dwell               : Real Microsecond,
                 emission-direction  : Radian}
  event GetSignal : Radar-Parameters
  var radar-parms : Radar-Parameters
  sort Radar-Response
                = { time-stamp       : Real Millisecond,
                    detect-object?   : Boolean,
                    object-location  : Range-Bearing}
  event RadarInfo : Radar-Response
  vars rc1, rc2 : Clock = 0
  vars c1, c2, c3 : Clock

  op P : MC-Radar-Parameters * MC-Radar-Response -> Boolean
  op Q : Radar-Parameters * Radar-Response -> Boolean
```

**Mission-Control-System =**

event RadarRequest : MC-Radar-Parameters
event RadarResult : MC-Radar-Response
event GetSignal : Radar-Parameters
event RadarInfo : Radar-Response
var parms : MC-Radar-Parameters
var rr : Radar-Response
op glue1 : MC-Radar-Parameters → Radar-Parameters
op glue2 : Radar-Response → MC-Radar-Response
vars mc1, mc2, c1, c2, c3, rc1, rc2 : Clock

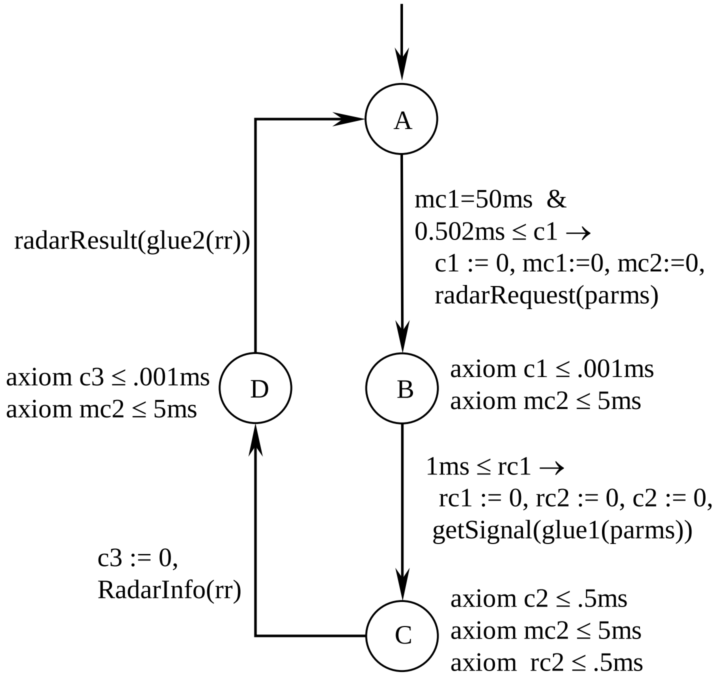


Figure 7: **Colimit of Figure 6**

```
  op glue1 : MC-Radar-Parameters -> Radar-Parameters
  op glue2 : Radar-Response -> MC-Radar-Response
  axiom glue is
    fa(in1:MC-Radar-Parameters, out1:Radar-Parameters,
       in2:Radar-Response, out2:MC-Radar-Response)
      (out1 = glue1(in1)
       & Q(out1,in2)
       & out2 = glue2(in2)
       => P(in1,out2))

 A = initial mode
      AB: step -> B
          mc1 = 50ms
          & 0.002ms + durRP <= c1 ->
            c1 := 0, mc1 := 0, mc2 := 0,
            RadarRequest(parms)
        end-step
     end-mode

 B = initial mode
      axiom c1 <= .001ms
      axiom mc2 <= 5ms
      BC: step -> C
         1ms <= rc1 ->
           c2 := 0, rc1 := 0, rc2 := 0,
           GetSignal(glue1(parms))
        end-step
     end-mode

 C = initial mode
      axiom c2 <= durRP
      axiom rc2 <= .5ms
      axiom mc2 <= 5ms
      CD: step -> D
         c3 := 0, RadarInfo(radar-result)
        end-step
     end-mode

 D = initial mode
      axiom c3 <= .001ms
      axiom mc2 <= 5ms
      DA: step -> A
         RadarResponse(glue2(radar-result))
        end-step
     end-mode

end-espec
```

### 4.2.4 Propagating Timing Constraints

When we formally compose the system, some timing incompatibilities arise. Here, the essence of the problem is that the radar can process requests no faster than 1000Hz, but the communication channel can send requests at rates up to 500KHz. They can work together, but we must eliminate those behaviors of the channel that are incompatible with the radar's service rate. We specified that the Mission Controller issues requests at 20Hz which is easily handled by the Radar. Suppose, for the sake of illustration, that we specified no minimum separation on the requests of the Mission Controller. In the following, we show how to set up a Constraint Satisfaction Problem (CSP) whose solution determines how to refine the composed system (by strengthening the guards and axioms) to achieve compatibility of all components of the system.

A Constraint Satisfaction problem is comprised of a set of variables, each with a given initial range of potential values, and a set of constraints that represent relationships between the variables. The goal of our CSP formulation is to find the tightest representation for the values of the variables that are consistent with the constraints. Another possible goal is to extract a feasible solution - an assignment of values to variables that satisfies the constraints. However, for this application, it is the runtime behavior of the system that does this extraction of a feasible solution.

The variables in the timed-Espec CSP are

- for each mode $m$, and each clock $c$, a variable representing the start time values for the clock in the mode: $m\text{-}c\text{-}st$;
- for each mode $m$, a variable representing the duration of the mode: $m\text{-}dur$.

In the following formulation of constraints, we are assuming for simplicity that each mode has one incoming transition and one outgoing transition. The generalization to multiple incoming or outgoing transitions is straightforward and not needed here. The constraints are as follows

1. $m\text{-}c\text{-}st \in [0..max]$ for each mode $m$ and clock $c$, where $max$ is some effectively infinite time bound.
2. $m\text{-}dur \in [0..max]$ for each mode $m$.
3. $m\text{-}dur \leq v$ for each mode $m$ with an axiom on clock $c$ of the form $c \leq v$.
4. $B\text{-}c\text{-}st = v$ for each transition $A \xrightarrow{g \vdash a} B$ in which $a$ has clock assignment `c := v`.
5. For each transition $A \xrightarrow{g \vdash a} B$ in which clock $c$ is not reset and $g$ has a constraint on $c$:
   (a) $B\text{-}c\text{-}st \leq v$ if the guard contains $c \leq v$.
   (b) $B\text{-}c\text{-}st = v$ if the guard contains $c = v$.
   (c) $B\text{-}c\text{-}st \geq v$ if the guard contains $c \geq v$.

6. $B\text{-}c\text{-}st = A\text{-}c\text{-}st + A\text{-}dur$ for each transition $A \xrightarrow{g \vdash a} B$ in which clock $c$ is not reset in $a$ and $g$ has no constraint on $c$.

7. For each transition $A \xrightarrow{g \vdash a} B$ in which $g$ has a constraint on clock $c$:

   (a) $A\text{-}c\text{-}st + A\text{-}dur \leq v$ if the guard contains $c \leq v$.

   (b) $A\text{-}c\text{-}st + A\text{-}dur = v$ if the guard contains $c = v$.

   (c) $A\text{-}c\text{-}st + A\text{-}dur \geq v$ if the guard contains $c \geq v$.

The constraint propagation process proceeds as follows. The rules above are used to interpret the timed Espec into a CSP. Each of the generated constraints are in definite clause form [18, 25] allowing a linear time propagation algorithm. Whenever a constraint is violated by current variable ranges, the solver performs the least refinement of one of the ranges so that the constraint is satisfied.

Figure 8 shows the propagation process. Column 1 lists the variables and the sucessive columns to the right represent the staged application of rules. That is, first rules 1 and 2 are applied to get initial ranges for the variables, shown in column 2. Then, rule 3 is applied, and so on. The essential inference here is that the activity represented by mode A must have a duration of at least 498$\mu$sec in order to maintain the minimum separation between requests that is required by the Radar unit.

If the constraint propagation reveals that there are no feasible values for some clock or some duration, then the system represented by the timed Espec is inconsistent. In other words the components are not composable. Otherwise, the inferred information constraints on the clock variables and mode durations help to refine the timed Espec in several ways.

1. Any inferred durations on a mode must be enforced (this is a refinement because we are subsetting the possible durations). A straightforward mechanism for enforcement is to add a new clock to the appropriate component so that it respects the duration constraint. In our hypothetical example above, the Mission Controller requires a clock to time activity A and restrict it to at least 498$\mu$sec, as in Figure 9.

2. Each guard in a transition can be simplified with respect to the inferred properties of the clock variables and durations.

3. Eliminate any clocks that are never checked in guards.

4. Add axioms to express tighter inferred bounds on clock values and durations at modes (effectively inverting rule 3).

5. Remove axioms that can be proved from others.

The results of applying constraint propagation and then these refinements are shown in Figure 9. This specification of the mission controller is the minimal refinement such that the composition with the radar unit and communication channel satisfies end-to-end timing constraints.

|  | Rules 1 & 2 | Rule 3 | Rule 4 | Rule 6 | Rule 7 | Rule 6 |
|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |
| A-mc-st | 0..max |  |  | 502..502 |  |  |
| A-c1-st | 0..max |  |  | 502..502 |  |  |
| A-c2-st | 0..max |  |  | 501..501 |  |  |
| A-c3-st | 0..max |  |  | 500..500 |  |  |
| A-rc1-st | 0..max |  |  | 501..501 |  |  |
| A-rc2-st | 0..max |  |  | 501..501 |  |  |
| A-dur | 0..max |  |  |  |  | 498..max |
|  |  |  |  |  |  |  |
| B-mc-st | 0..max |  | 0..0 |  |  |  |
| B-c1-st | 0..max |  | 0..0 |  |  |  |
| B-c2-st | 0..max |  |  |  |  | 999..max |
| B-c3-st | 0..max |  |  |  |  | 998..max |
| B-rc1-st | 0..max |  |  |  | 999..max |  |
| B-rc2-st | 0..max |  |  |  |  | 999..max |
| B-dur | 0..max | 1..1 |  |  |  |  |
|  |  |  |  |  |  |  |
| C-mc-st | 0..max |  |  | 1..1 |  |  |
| C-c1-st | 0..max |  |  | 1..1 |  |  |
| C-c2-st | 0..max |  | 0..0 |  |  |  |
| C-c3-st | 0..max |  |  |  |  | 999..max |
| C-rc1-st | 0..max |  | 0..0 |  |  |  |
| C-rc2-st | 0..max |  | 0..0 |  |  |  |
| C-dur | 0..max | 500..500 |  |  |  |  |
|  |  |  |  |  |  |  |
| D-mc-st | 0..max |  |  | 501..501 |  |  |
| D-c1-st | 0..max |  |  | 501..501 |  |  |
| D-c2-st | 0..max |  |  | 500..500 |  |  |
| D-c3-st | 0..max |  | 0..0 |  |  |  |
| D-rc1-st | 0..max |  |  | 500..500 |  |  |
| D-rc2-st | 0..max |  |  | 500..500 |  |  |
| D-dur | 0..max | 1..1 |  |  |  |  |

Figure 8: Propagation of Timing Constraints

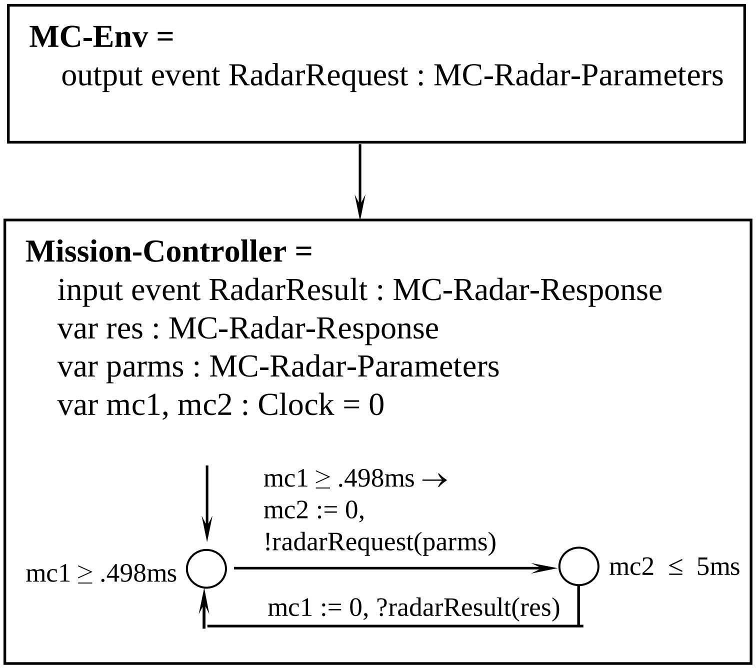

MC-Env =
output event RadarRequest : MC-Radar-Parameters
Mission-Controller =
input event RadarResult : MC-Radar-Response
var res : MC-Radar-Response
var parms : MC-Radar-Parameters
var mc1, mc2 : Clock = 0
mc1 ≥ .498ms →
mc2 := 0,
!radarRequest(parms)
mc1 ≥ .498ms
mc2 ≤ 5ms
mc1 := 0, ?radarResult(res)

Figure 9: Mission Controller with minimum acceptable request separation times

The main message of this section is that after composition, we can automatically and scalably eliminate incompatible behaviors of the components with respect to timing properties. If the propagation process does not determine inconsistency, then we know that the components are composable according to the architecture.

### 4.2.5 Glue-Code Generation

The colimit `Mission-Control-System` is an Espec for the joint action of the Mission Controller, CC mechanism, and Radar unit. A fragment of it is shown below to point out the opportunity for glue code generation. Notice that the datatypes `MC-Radar-Parameters` and `Radar-Parameters` differ – they have the same fields, but not in the same order. Similarly, the datatypes `MC-Radar-Response` and `Radar-Response` differ – one represents the location of a detected object in latitute-longitude pairs, and the other represents it in relative polar coordinates.

```
Mission-Control-System = espec
    sort MC-Radar-Parameters
            =  { gain                : Real,
                 frequency           : Real Microsecond,
                 dwell               : Real Hertz,
                 emission-direction  : Real }
    sort MC-Radar-Response
                = { time-stamp       : Real Millisecond,
                    detect-object?   : Boolean,
                    object-location  : Lat-Long}
    sort Radar-Parameters
            =  { gain                : Real,
                 frequency           : Real Hertz,
                 dwell               : Real Microsecond,
                 emission-direction  : Radian}
    sort Radar-Response
                = { time-stamp       : Real Millisecond,
                    detect-object?   : Boolean,
                    object-location  : Range-Bearing}

  op glue1 : MC-Radar-Parameters -> Radar-Parameters
  op glue2 : Radar-Response -> MC-Radar-Response
  axiom glue is
    fa(in1:MC-Radar-Parameters, out1:Radar-Parameters,
       in2:Radar-Response, out2:MC-Radar-Response)
      (out1 = glue1(in1)
       & IsValidSignal (out1,in2)
       & out2 = glue2(in2)
       => MC-IsValidSignal (in1,out2))
  ...
end-espec
```

A glue-code generator uses the `glue` axiom to generate appropriate definitions for `glue1` and `glue2`. The technique is to unskolemize the two underspecified operators in the `glue` axiom, and then to perform constructive inference to find witnesses for the existentials [19, 4]. In this case the following definitions provide a valid refinement of MC-plus-CC-plus-RADAR with operational glue code.

```
Mission-Control-System = espec
  ...
  op glue1 : MC-Radar-Parameters -> Radar-Parameters
  def glue1(<g,f,d,e>) = <g,d,f,e>
  op glue2 : Radar-Response -> MC-Radar-Response
  def glue2(<ts,do,ol-ll>) = <ts,do,convert-polar-to-lat-long(ol-ll)>
  ...
end-espec
```

## 5 Concluding remarks and future work

Epoxi builds on concepts from Specware [24], overcoming its bias towards generating functional code by supporting behavioral specifications and the generation of complex systems. Epoxi also builds on previous efforts to model behavior logically (e.g. [8, 12]) by defining a formal notion of composition (via colimit) and refinement (via morphisms). Epoxi represents an advance on previous refinement methods, such as VDM and B, in a variety of ways. The categorical foundations support controlled sharing of substructure, a uniform approach to datatype refinement, and greater automated support for composition and refinement.

We believe that Evolving Specifications and their implementation in Epoxi represents groundbreaking work in systems design. No other formalism known to us caters for the full range of (1) precise specification of logical, temporal, and behavioral properties, (2) precise specification of required and offered services of components and connectors, (3) precise and automatable composition (via diagrams of Especs), (4) design-time verification of the compliance of a composition (by verifying that required properties are satisfied by a component's environment), (5) precise and mechanizable refinement of systems in a way that preserves specified properties and produces correct-by-construction code. Moreover, as discussed in Section 3.11, the integration of both composition and refinement in Epoxi points out the need for further distinctions in the concepts of software architecture, for example, to support the fact that environmental assumptions refine contravariantly.

The Epoxi framework has allowed us to explore precise means for assessing composability. This project developed techniques for formalizing (1) the compliance of a software artifact with its specification at several levels of granularity, and (2) the compliance of a component's environment with the services, behaviors, and properties that the component requires. The primary measure of composability is the existence of a morphism from each required-service specification of a system to the offered-service specifications of its environment. We also developed several techniques for ensuring composability in the presence of semantic mismatches between parts of a system. When there is a gap in the logical properties at an interface, we developed techniques for generating glue code when translation is possible. When there are race-condition-like timing problems between components, we developed constraint propagation techniques that eliminate uncoordinated behaviors while preserving logical properties.

The Epoxi concepts has been demonstrated on a rich variety of systems problems. Moreover, the foundational results obtained in this project have given rise to follow-on efforts in a variety of directions. First, the Planware system, inspired by Especs, is being applied to the automatic generation of high-performance scheduling codes for Theater Battle Management and other defense applications. Second, under NSA support we are extending Especs to handle object-oriented modeling features, essentially by allowing dynamic binding of parameters. This project is building generators of C and Java code from Especs. Third, Especs are being used to specify and compose features of authentication protocols. Fourth, Especs are being used as the basis for exploring the development and application of game theory to the problems of managing distributed agent systems. Other projects are extending Especs to support the development of hybrid embedded systems and the automatic robustification of Java systems.

# References

[1] Abadi, M., and Lamport, L. Conjoining specifications. *ACM Transactions on Programming Languages and Systems 13*, 3 (1995), 507–534.

[2] Alur, R., and Dill, D. A theory of timed automata. *Theoretical Computer Science 126* (1994), 183–235.

[3] Becker, M., Gilham, L., and Smith, D. R. Planware II: Synthesis of schedulers for complex resource systems. Tech. rep., Kestrel Technology, 2003.

[4] Burstein, M., McDermott, D., Smith, D., and Westfold, S. Derivation of glue code for agent interoperation. *Journal of Autonomous Agents and Multi-Agent Systems 6* (2003), 265–286.

[5] Durgin, N., Mitchell, J., and Pavlovic, D. A compositional logic for proving security properties of protocols. *J. of Comp. Security 11*, 4 (2004), 677–721.

[6] Errington, L. Notes on diagrams and state. Tech. rep., Kestrel Institute, 2000.

[7] Goguen, J. A., and Burstall, R. M. Institutions: Abstract model theory for computer science. Tech. Rep. CSLI-85-30, Stanford University, 1985.

[8] Gurevich, Y. Evolving algebra 1993: Lipari guide. In *Specification and Validation Methods*, E. Boerger, Ed. Oxford University Press, 1995, pp. 9–36.

[9] J. Hill, et al. Tinyos: An operating system for sensor networks. Tech. rep., Dept of EECS, Univ. California at Berkeley, 2000.

[10] J.L.Fiadeiro, and T.Maibaum. Interconnecting formalisms: supporting modularity, reuse and incrementality. In *Proc. 3rd Symposium on the Foundations of Software Engineering* (1995), G. Kaiser, Ed., ACM Press, pp. 72–80.

[11] MacLane, S. *Categories for the Working Mathematician*, vol. 5 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1971.

[12] Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1992.

[13] Pavlovic, D. Semantics of first order parametric specifications. In *Formal Methods '99* (1999), J. Woodcock and J. Wing, Eds., vol. 1708 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 155–172.

[14] Pavlovic, D., Pepper, P., and Smith, D. R. Colimits for concurrent collectors. In *Verification: Theory and Practice: Festschrift for Zohar Manna* (2003), N. Dershowitz, Ed., LNCS 2772, pp. 568–597.

[15] Pavlovic, D., and Smith, D. R. Composition and refinement of behavioral specifications. In *Proceedings of Automated Software Engineering Conference* (2001), IEEE Computer Society Press, pp. 157–165.

[16] Pavlovic, D., and Smith, D. R. System construction via evolving specifications. In *Complex and Dynamic Systems Architectures (CDSA 2001)* (2001).

[17] Pavlovic, D., and Smith, D. R. Guarded transitions in evolving specifications. In *Proceedings of Algebraic Methods in Software Technology (AMAST)* (2002), H. Kirchner and C. Ringeissen, Eds., Springer-Verlag LNCS, pp. 411–425.

[18] Rehof, J., and Mogenson, T. Tractable constraints in finite semilattices. *Science of Computer Programming 35* (1999), 191–221.

[19] Smith, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.

[20] Smith, D. R. Mechanizing the development of software. In *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, M. Broy and R. Steinbrueggen, Eds. IOS Press, Amsterdam, 1999, pp. 251–292.

[21] Smith, D. R. Toward formal development of embedded systems. Tech. rep., Kestrel Technology, 2002.

[22] Smith, D. R., Parra, E. A., and Westfold, S. J. Synthesis of planning and scheduling software. In *Advanced Planning Technology* (1996), A. Tate, Ed., AAAI Press, Menlo Park, pp. 226–234.

[23] Srinivas, Y. V. Refinements of parameterized algebraic specifications. In *Algorithmic Languages and Calculi*, R. Bird and L. Meertens, Eds. Chapman & Hall, London, 1997.

[24] Srinivas, Y. V., and Jüllig, R. Specware: Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399–422.

[25] Westfold, S., and Smith, D. Synthesis of efficient constraint satisfaction programs. *Knowledge Engineering Review 16*, 1 (2001), 69–84. (Special Issue on AI and OR).